Array

Items:$x_0, x_1, \ldots, x_{n-1}$

Build(): given an iterable X, build sequence from items in X

Len(): return n

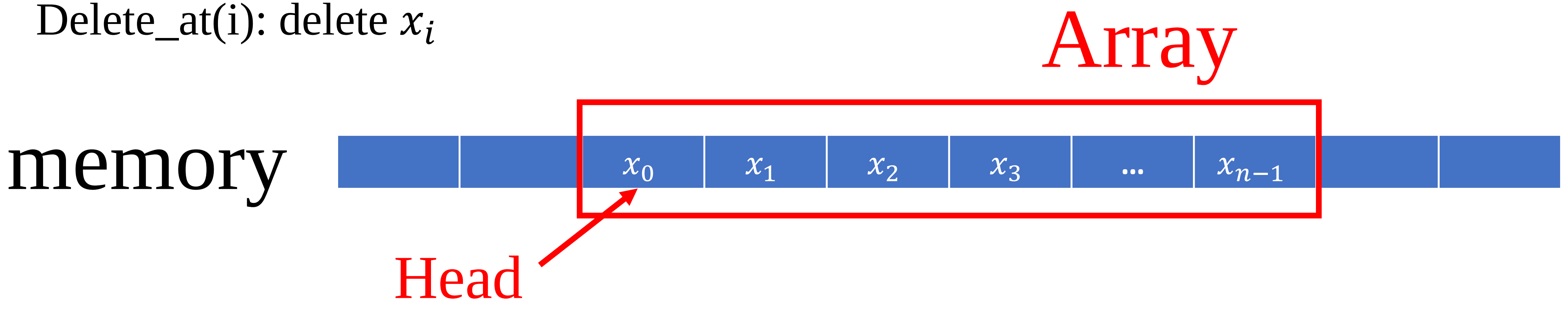
Find(k): return the stored item with key k

Get_at(i): retrun x_i

Set_at(I, x): set x_i $to\ x$

Insert_at(i, x): make x the new x_i

Delete_at(i): delete x_i

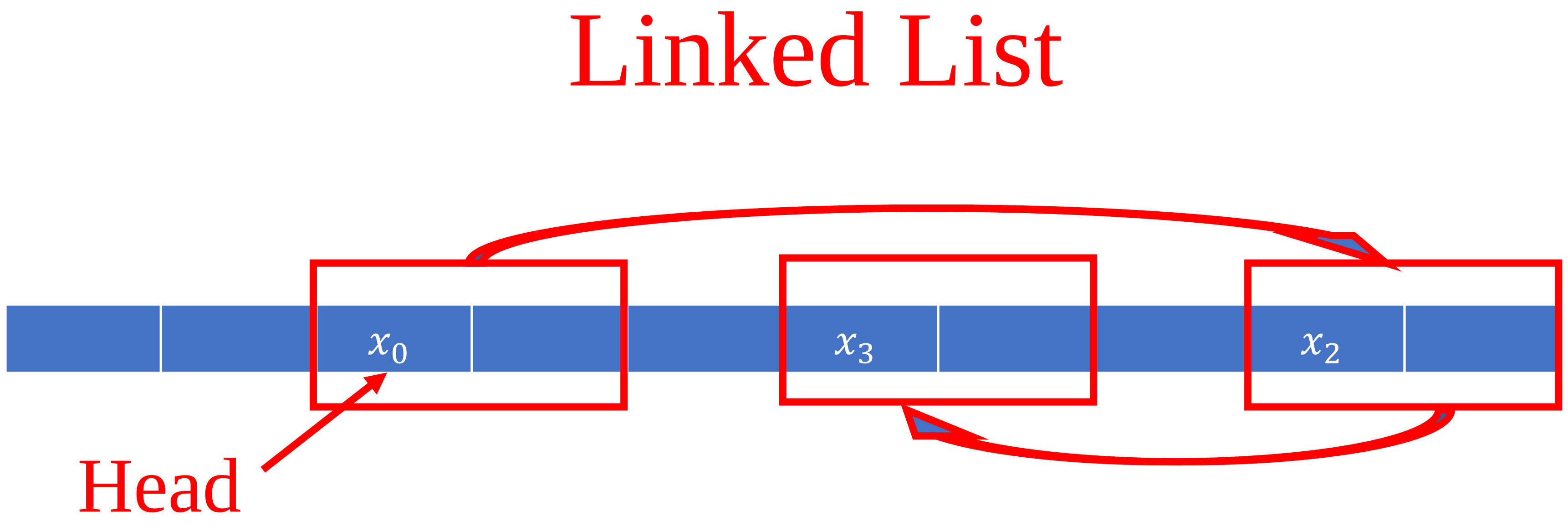


Array

Items:$x_0, x_1, \dots, x_{n-1}$	
Build(): given an iterable X, build sequence from items in X	O(n)
Len(): return n	O(1)
Find(k): return the stored item with key k	O(n)
Get_at(i): retrun x_i	O(1)
Set_at(I, x): set $x_i\ to\ x$	O(1)
Insert_at(i, x): make x the new x_i	O(n)
Delete_at(i): delete x_i	O(n)

Linked List

- Build(X)
- Get_at(i)
- Set_at(i, x)
- Insert_at(i, x)
- Delete_at(x)
- Insert_first(x)
- Delete_first()
- insert_last(x)
- Delete_last()



memory

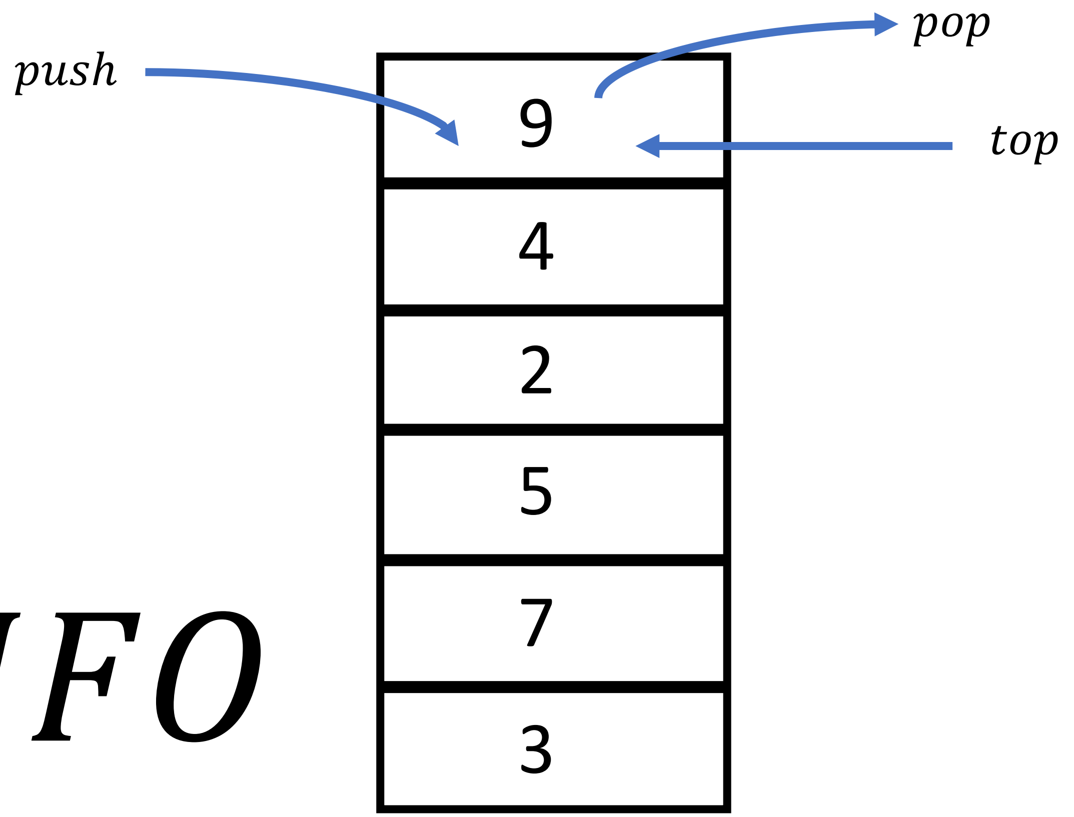
Linked list

• Build(X)	O(n)
• Get_at(i)	O(n)
• Set_at(i, x)	O(n)
• Insert_at(i, x)	O(n)
• Delete_at(x)	O(n)
• Insert_first(x)	O(1)
• Delete_first()	O(1)
• insert_last(x)	O(n)
• Delete_last()	O(n)

Stack

- push(value)
- Pop()
- Top()
- isEmpty()

$LIFO$



Stack

Let’s say we had a program like this:

```
main() {
   function1();
   return;
}
```

```
function1() {
   function2();
   return;
}
```

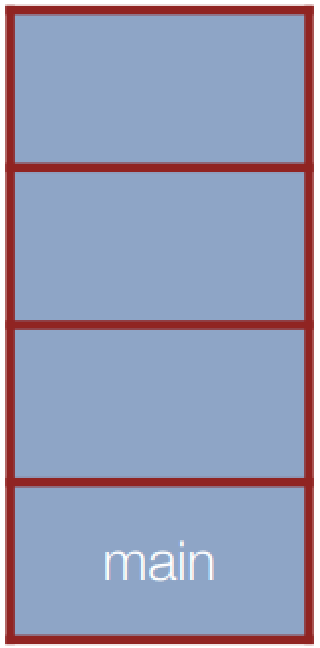
```
function2() {
   function3();
   return;
}
```

What happens to the state of the system as this program runs?

```
main() {
    function1();
    return;
}
```

```
function1() {
    function2();
    return;
}
```

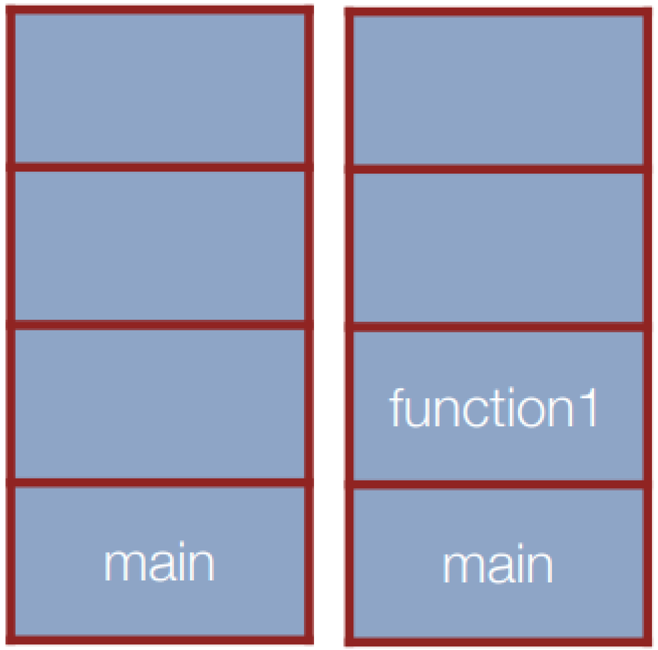
```
function2() {
    function3();
    return;
}
```



```
main() {
    function1();
    return;
}
```

```
function1() {
    function2();
    return;
}
```

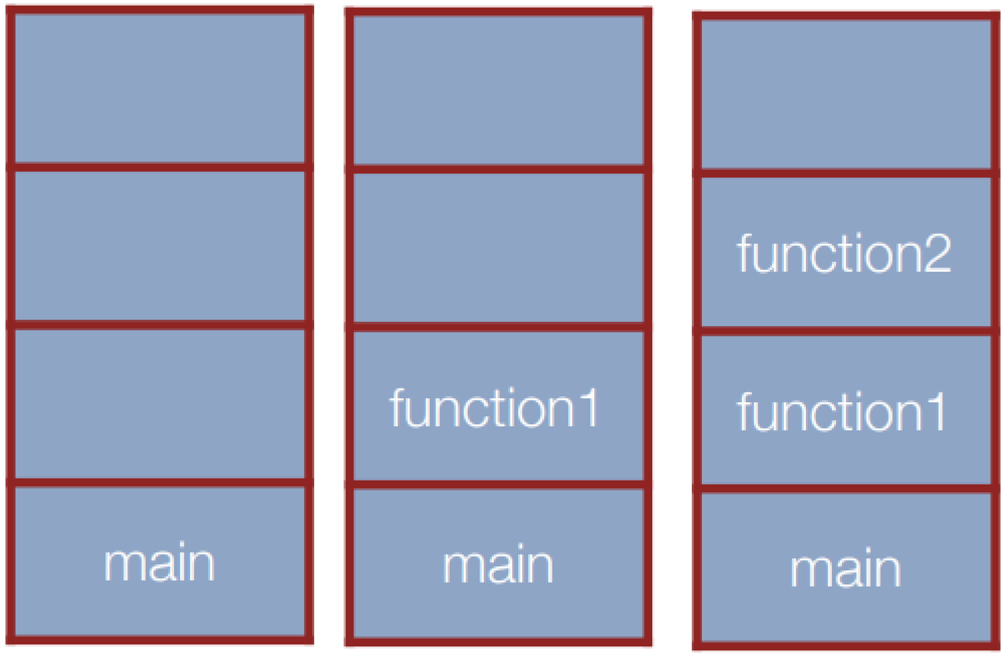
```
function2() {
    function3();
    return;
}
```



```
main() {
    function1();
    return;
}
```

```
function1() {
    function2();
    return;
}
```

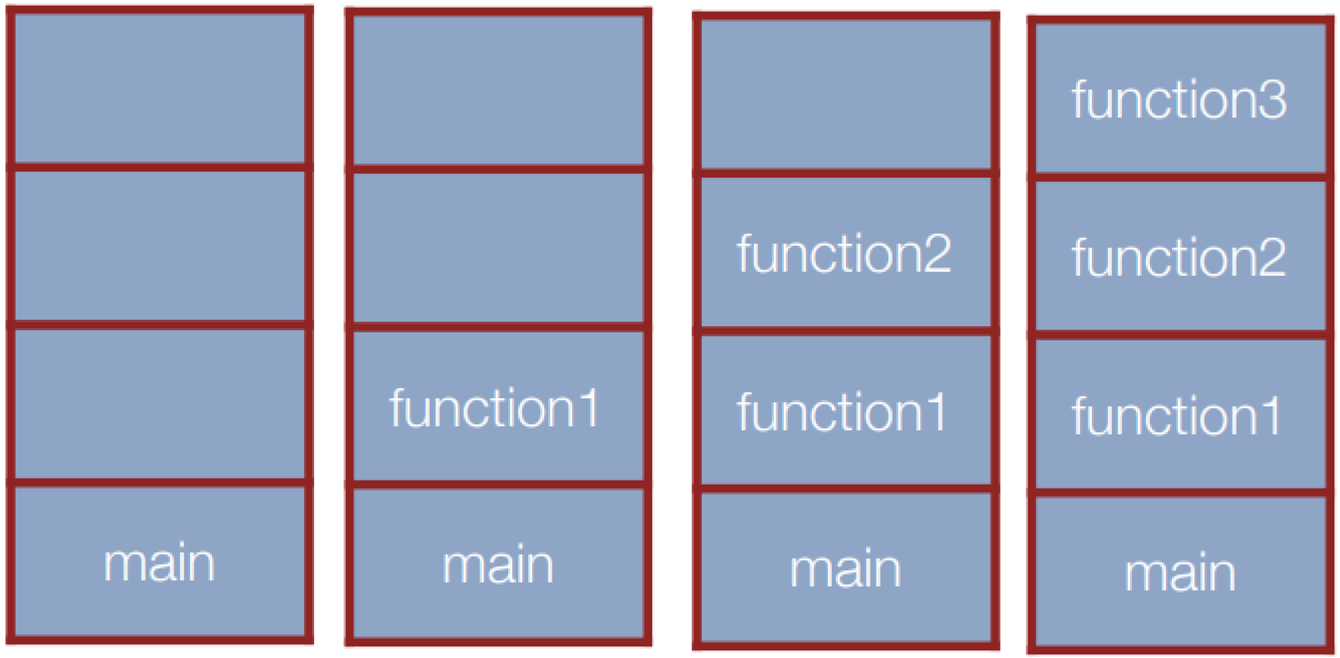
```
function2() {
    function3();
    return;
}
```



```
main() {
    function1();
    return;
}
```

```
function1() {
    function2();
    return;
}
```

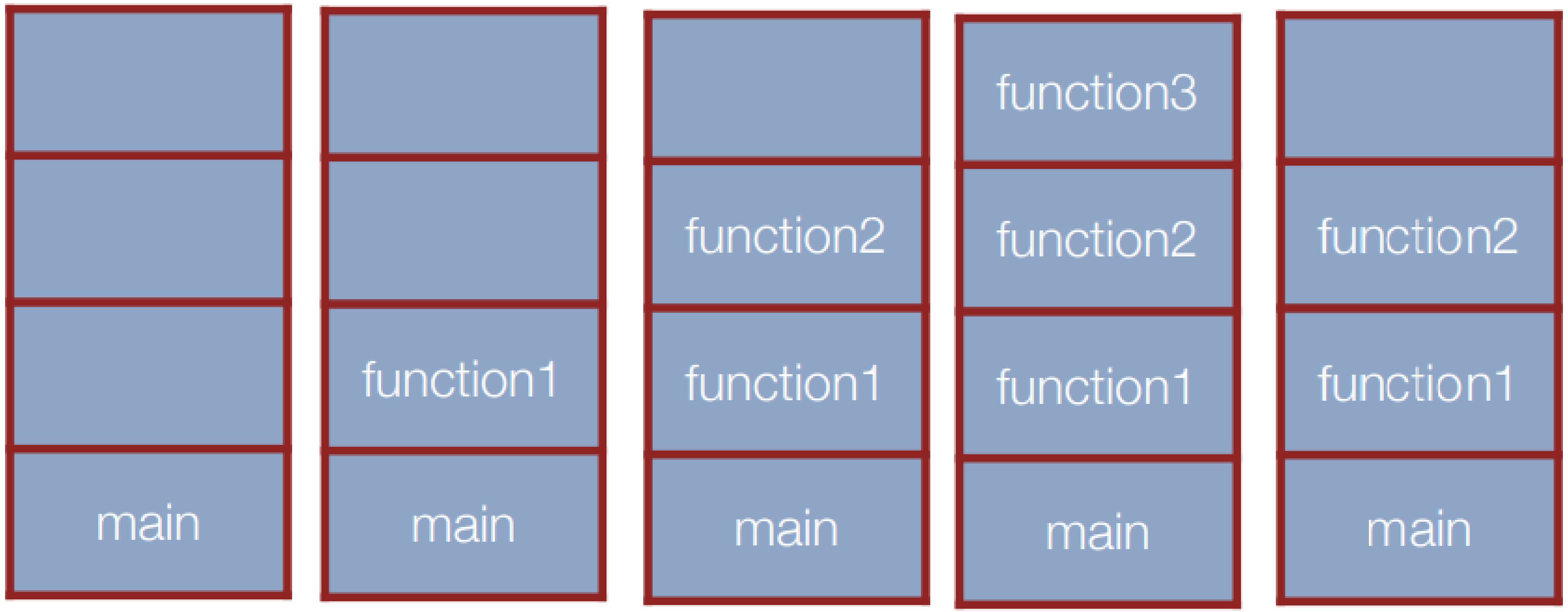
```
function2() {
    function3();
    return;
}
```



```
main() {
    function1();
    return;
}
```

```
function1() {
    function2();
    return;
}
```

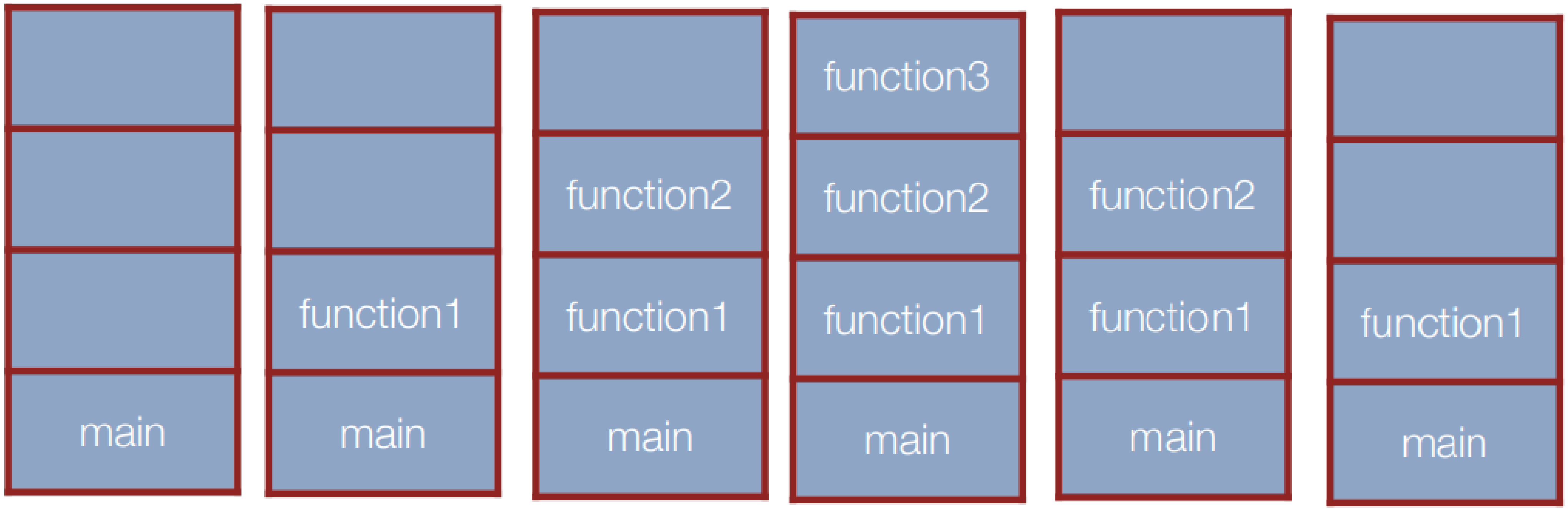
```
function2() {
    function3();
    return;
}
```



```
main() {
    function1();
    return;
}
```

```
function1() {
    function2();
    return;
}
```

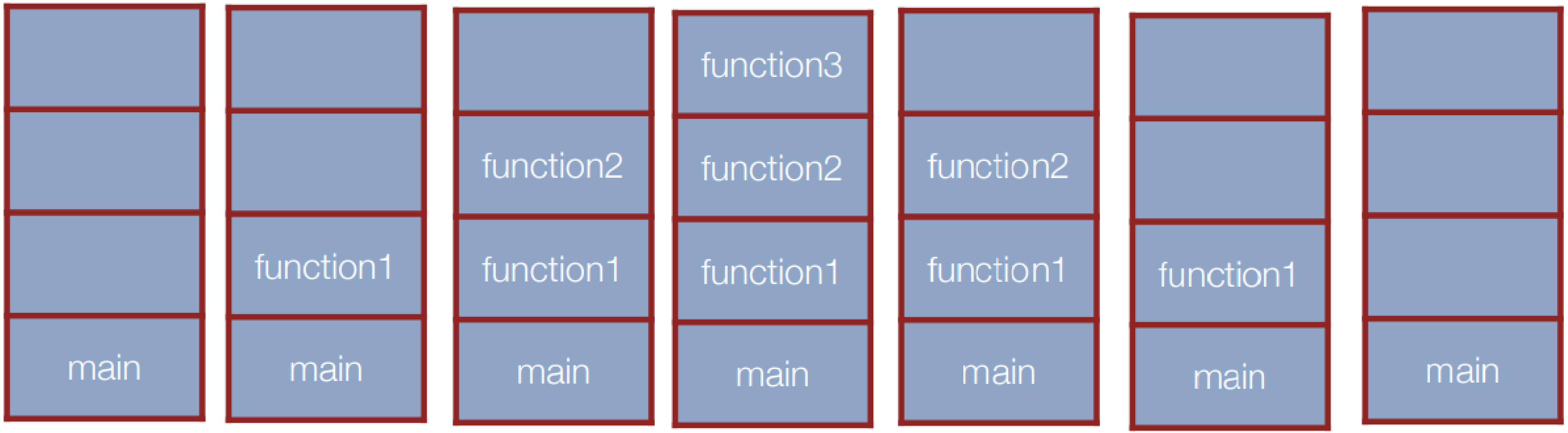
```
function2() {
    function3();
    return;
}
```



```
main() {
    function1();
    return;
}
```

```
function1() {
    function2();
    return;
}
```

```
function2() {
    function3();
    return;
}
```



Advanced Stack Example

$$5 * 4 - 8 /2 + 9$$

Use “postfit” notation

Advanced Stack Example

$$5 * 4 - 8 /2 + 9$$

Use “postfit” notation

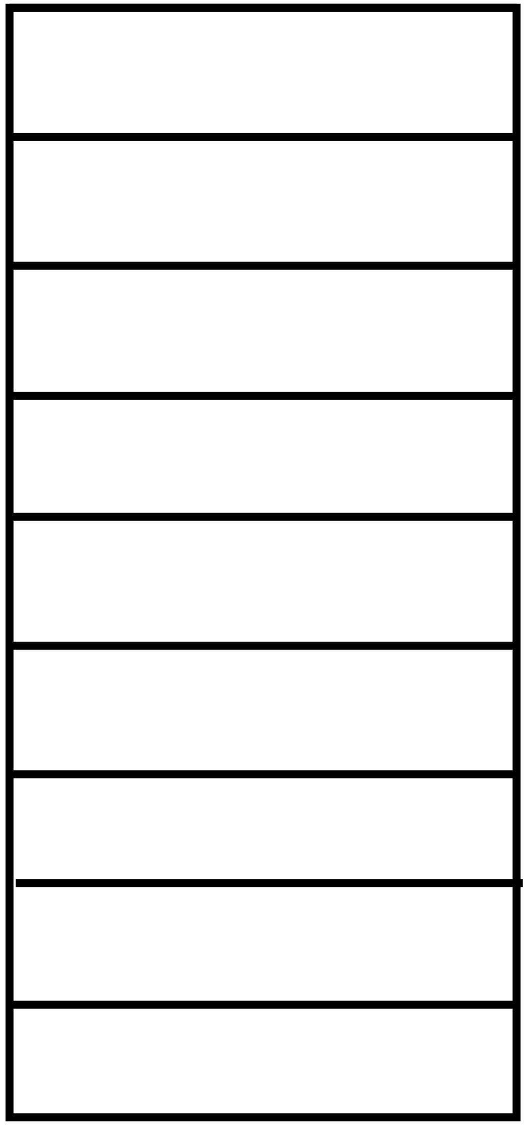
$$5\ 4\ \ *\ 8\ \ 2\ / - 9 +$$

Advanced Stack Example

$$5 * 4 - 8\,/2 + 9$$

Use “postfit” notation

$$5\;4\;\;*\;8\;\;2\;/{-9}+$$

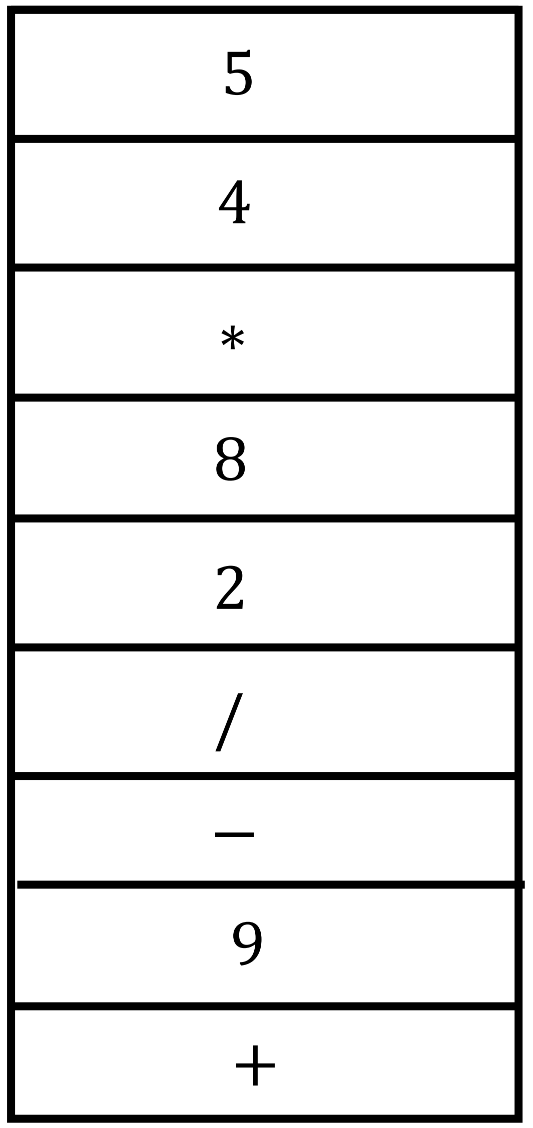


Advanced Stack Example

$$5 * 4 - 8 /2 + 9$$

Use “postfit” notation

$$5\ 4\ *\ 8\ 2\ /-9 +$$

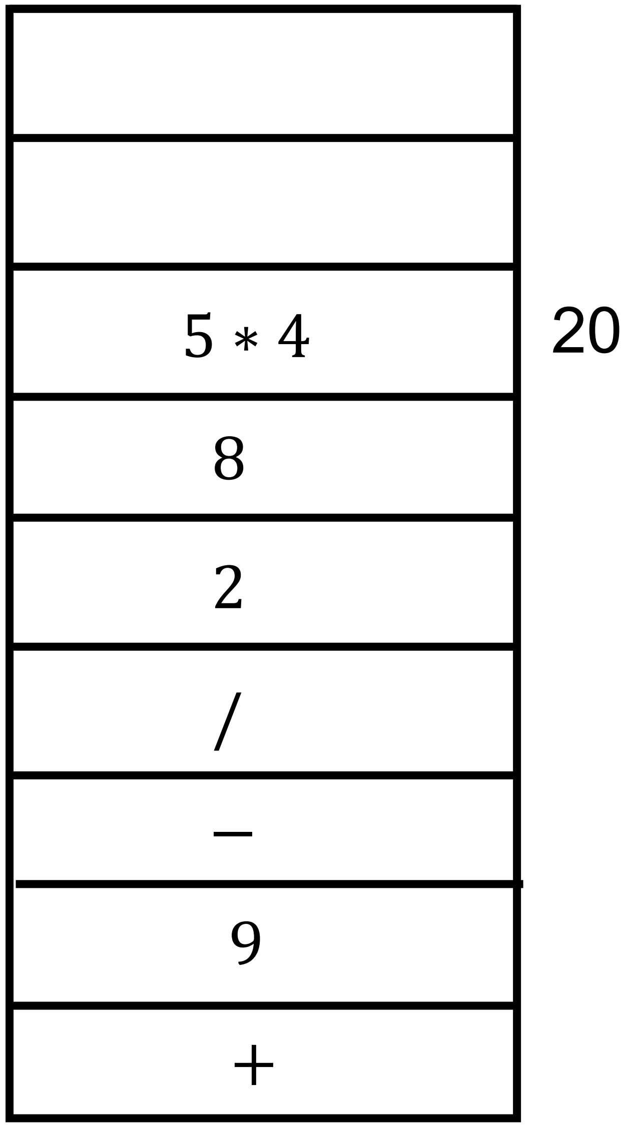


Advanced Stack Example

$$5 * 4 - 8 /2 + 9$$

Use “postfit” notation

$$5\ 4\ * 8\ 2\ /-9 +$$

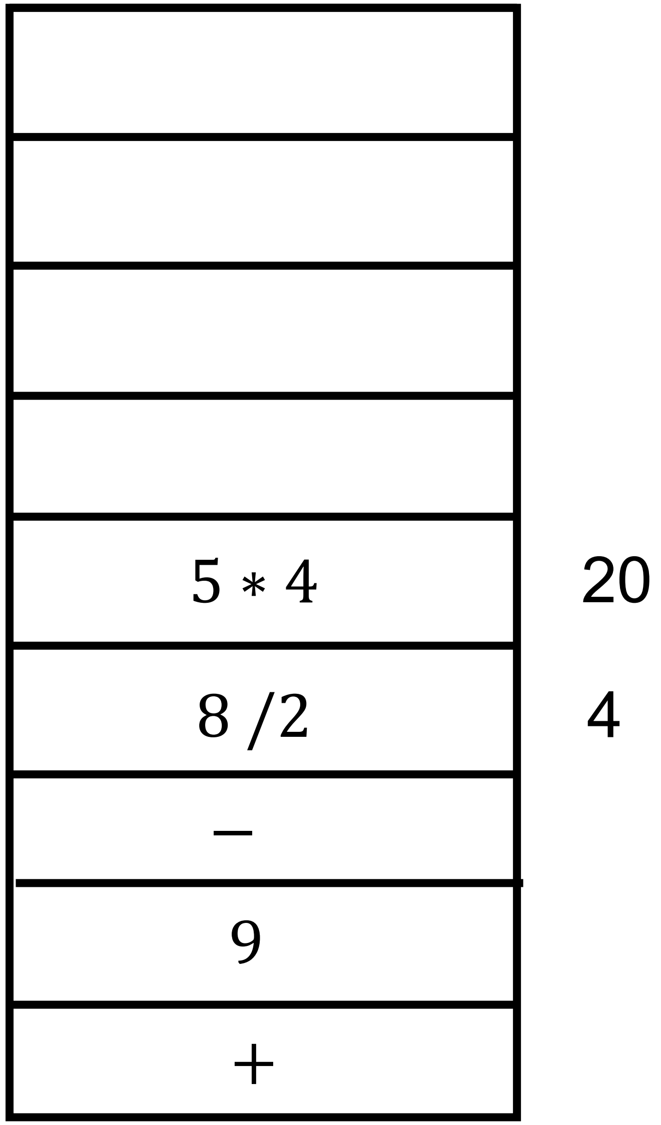


Advanced Stack Example

$$5 * 4 - 8 /2 + 9$$

Use “postfit” notation

$$5\ 4\ * 8\ 2\ /-9+$$

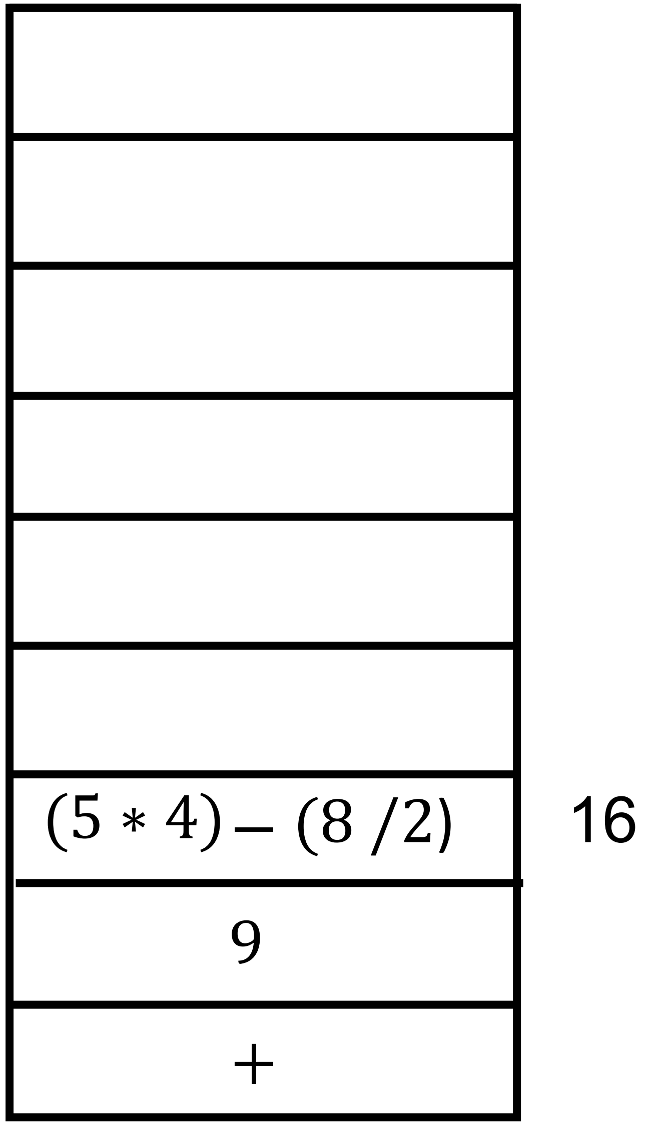


Advanced Stack Example

$$5 * 4 - 8 / 2 + 9$$

Use “postfit” notation

$$5\ 4\ *\ 8\ 2\ /\ -\ 9\ +$$

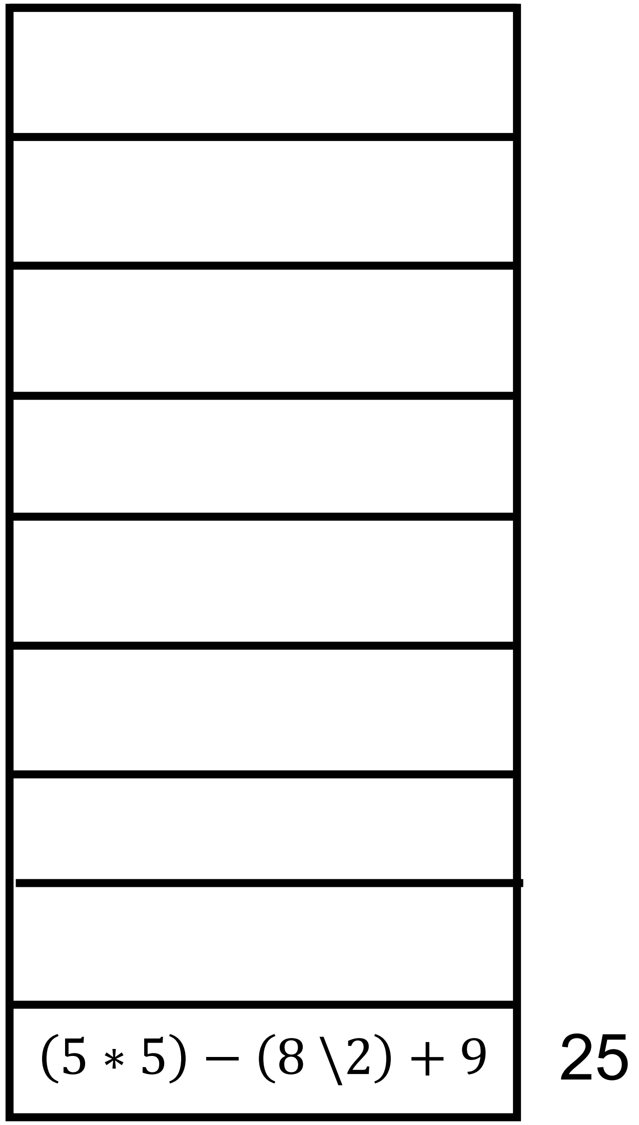


Advanced Stack Example

$$5 * 4 - 8\ /2 + 9$$

Use “postfit” notation

$$5\ 4\ \ * 8\ \ 2\ /-9 +$$

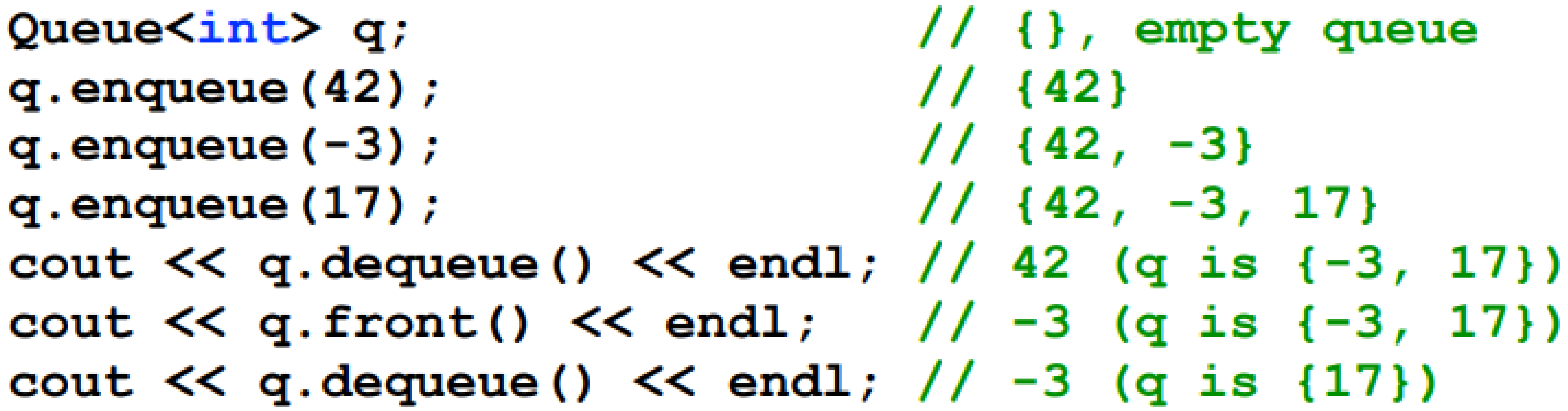


Queue

- enqueue(value) (or add(value))
- dequeue() (or remove())
- front() (or peek())
- isEmpty()

Queue

- enqueue(value) (or add(value))
- dequeue() (or remove())
- front() (or peek())
- isEmpty()



```
Queue<int> q;                      // {}, empty queue
q.enqueue(42);                     // {42}
q.enqueue(-3);                     // {42, -3}
q.enqueue(17);                     // {42, -3, 17}
cout << q.dequeue() << endl; // 42 (q is {-3, 17})
cout << q.front() << endl;   // -3 (q is {-3, 17})
cout << q.dequeue() << endl; // -3 (q is {17})
```

Queue Mystery

What is the output of the following code?

```
Queue<int> queue;
// produce: {1, 2, 3, 4, 5, 6}
for (int i = 1; i <= 6; i++) {
    queue.enqueue(i);
}
for (int i = 0; i < queue.size(); i++) {
    cout << queue.dequeue() << " ";
}
cout << queue.toString() << "  size " << queue.size() << endl;
```

```
A. 1 2 3 4 5 6 {} size 0
B. 1 2 3 {4,5,6} size 3
C. 1 2 3 4 5 6 {1,2,3,4,5,6} size 6
D. none of the above
```

Queue Mystery

What is the output of the following code?

```
Queue<int> queue;
// produce: {1, 2, 3, 4, 5, 6}
for (int i = 1; i <= 6; i++) {
    queue.enqueue(i);
}
for (int i = 0; i < queue.size(); i++) {
    cout << queue.dequeue() << " ";
}
cout << queue.toString() << "  size " << queue.size() << endl;
```

Changes during the loop! Be careful!!

```
A. 1 2 3 4 5 6 {} size 0
[✓ B. 1 2 3 {4,5,6} size 3]
C. 1 2 3 4 5 6 {1,2,3,4,5,6} size 6
D. none of the above
```