Implementing Abstractions

Part Two

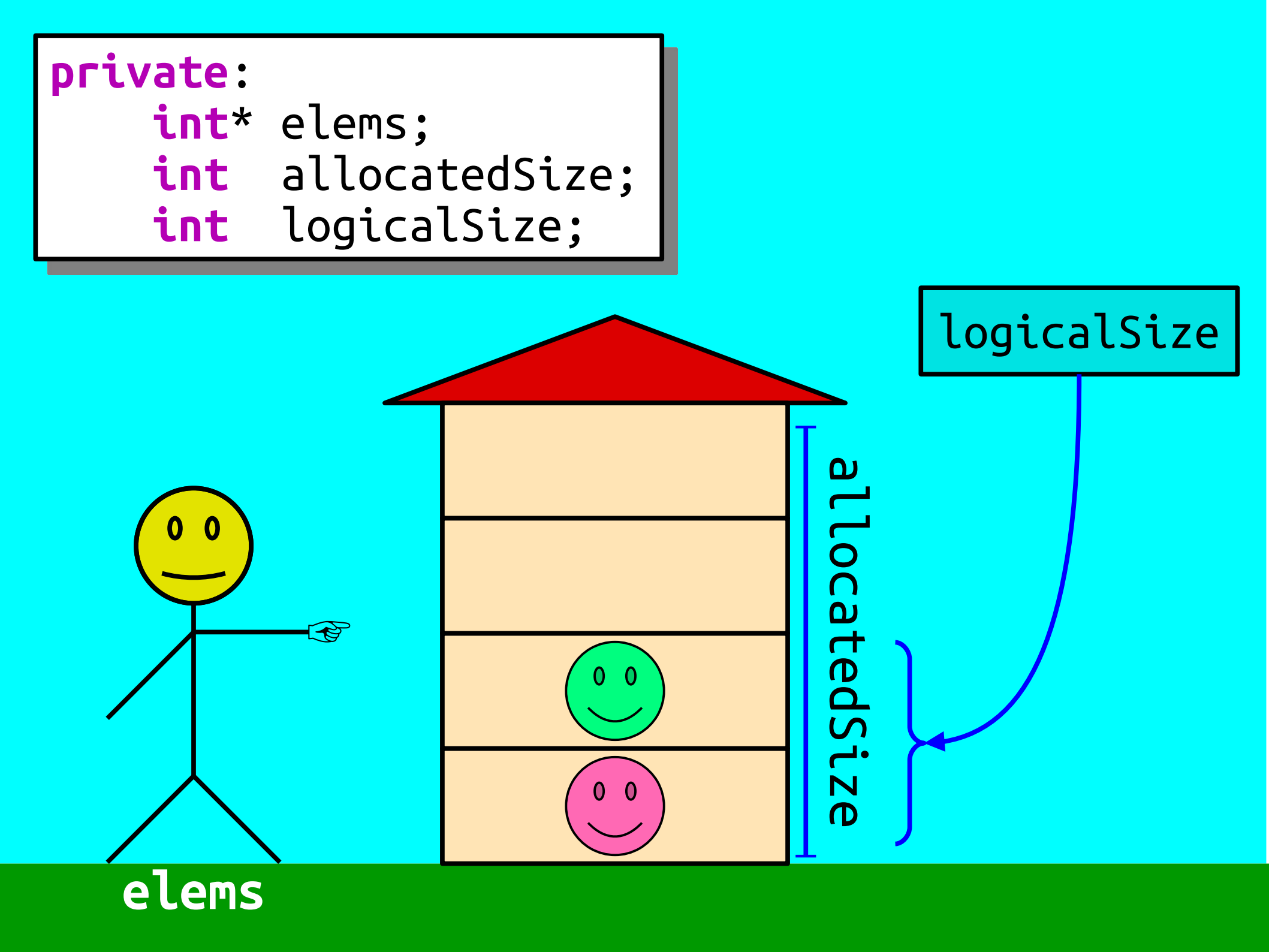
Previously, on CS106B...

```
class OurStack {
public:
    OurStack();

    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```

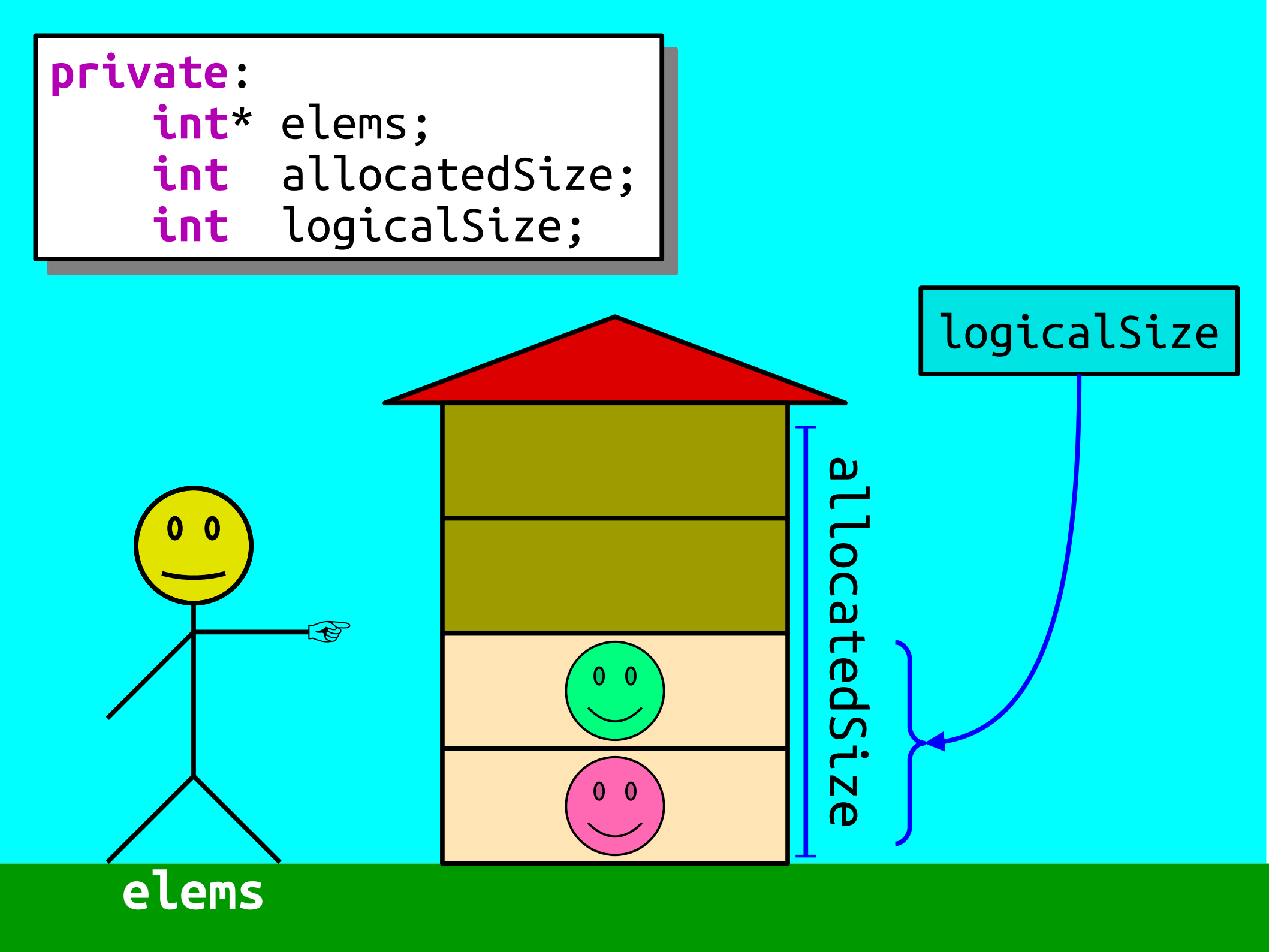
```
class OurStack {
public:
    OurStack();

    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```

```
class OurStack {
public:
    OurStack();

    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```

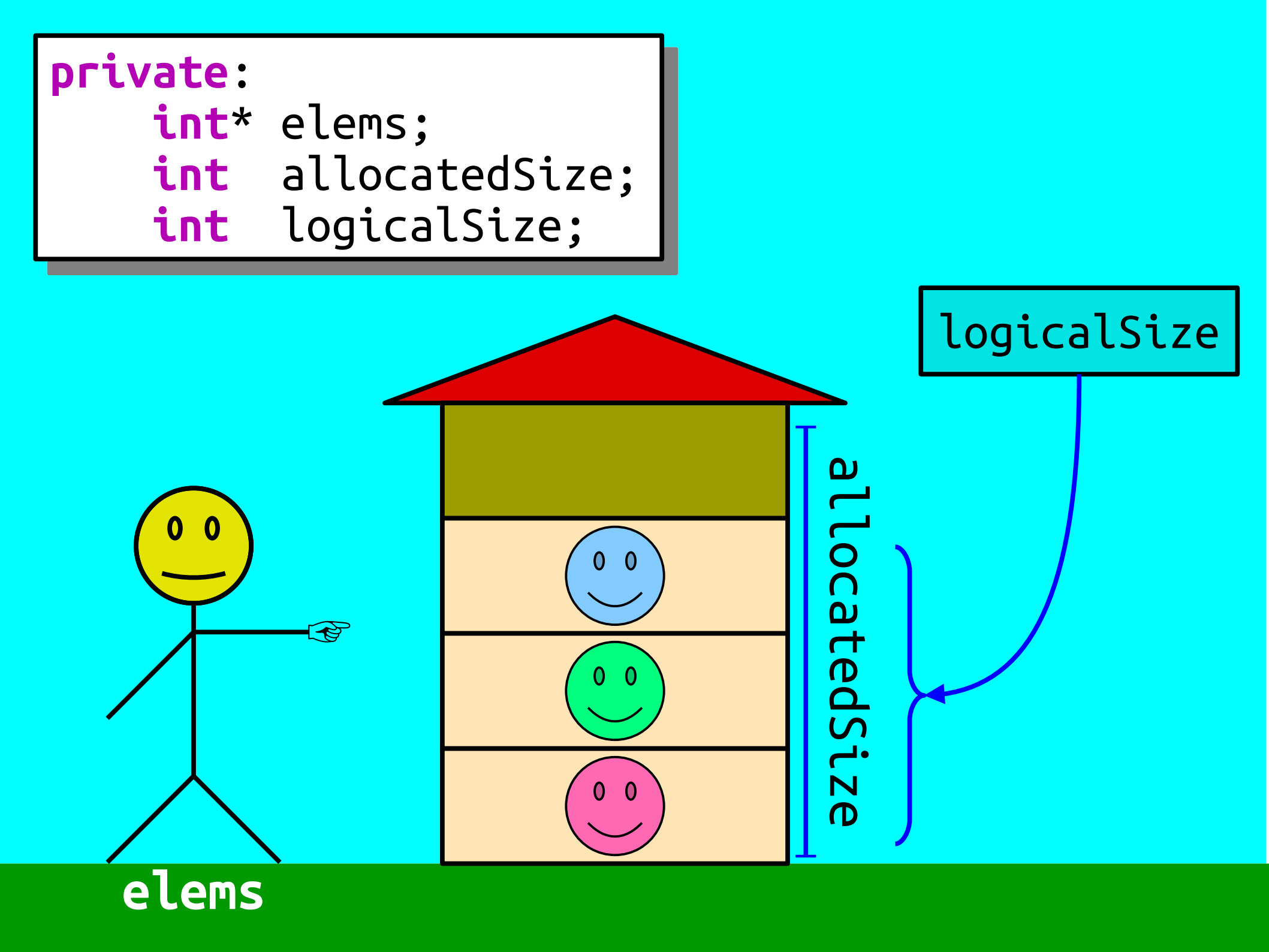
```
private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
```



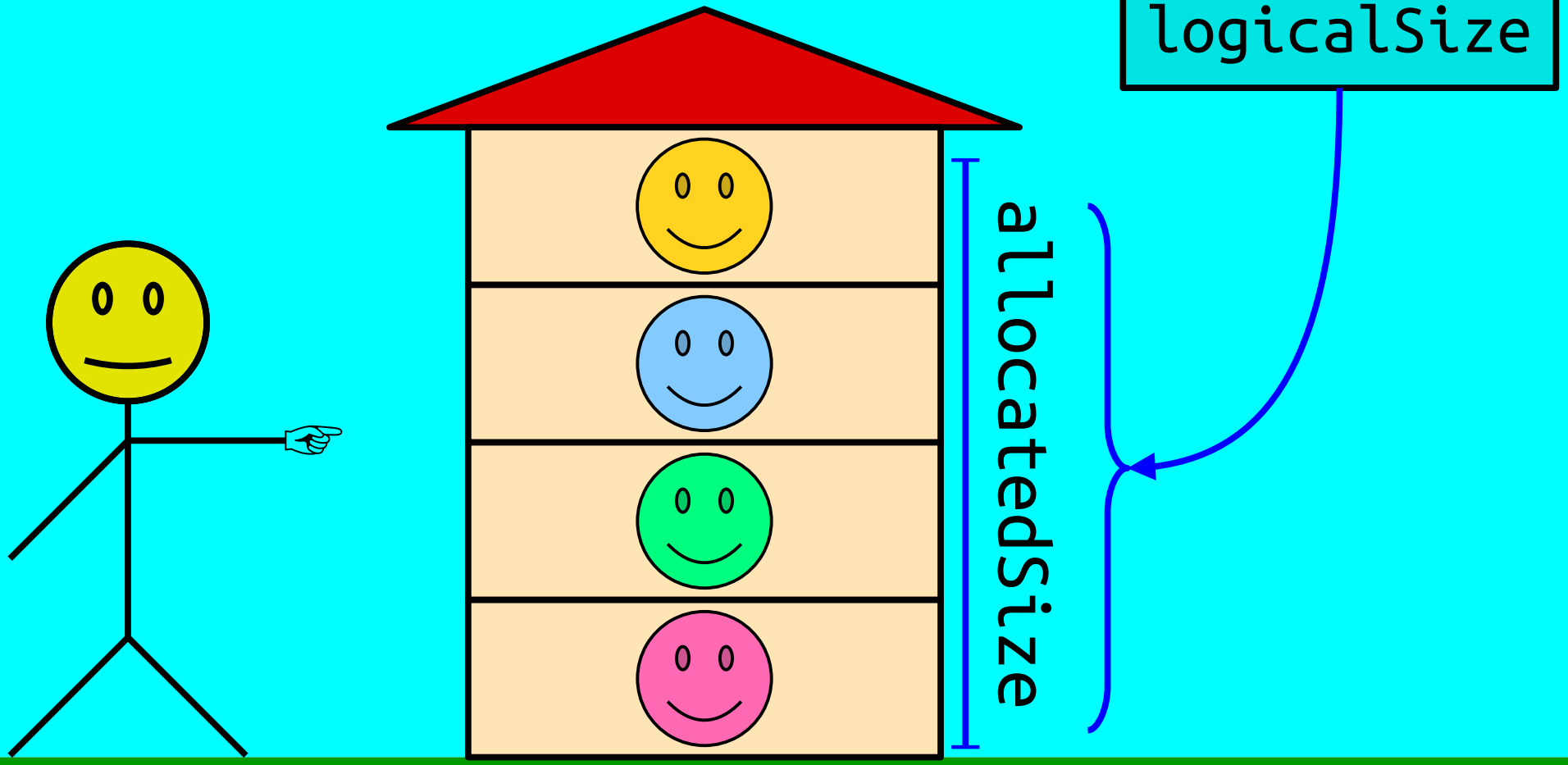
```
private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
```



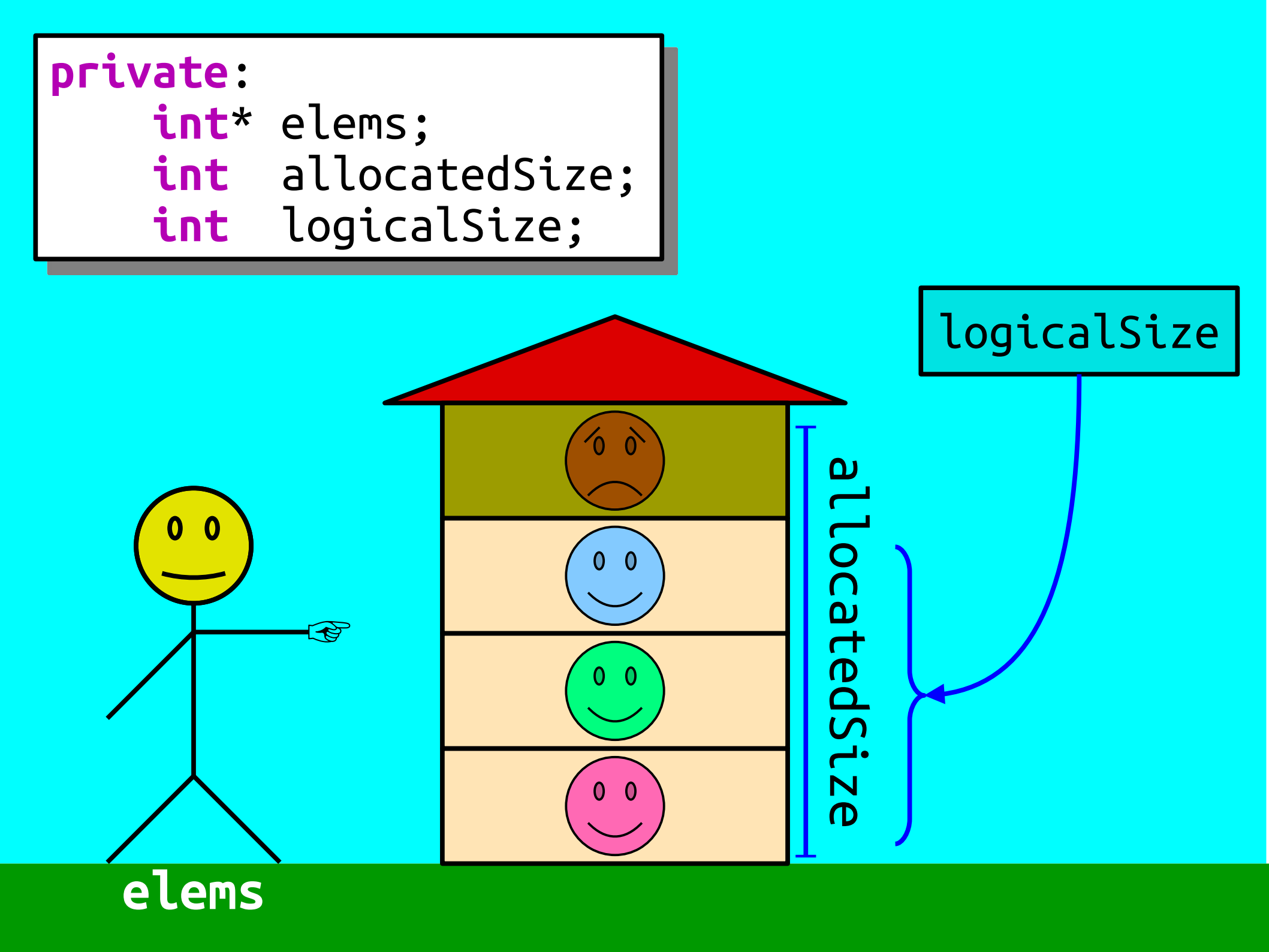
```
private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
```



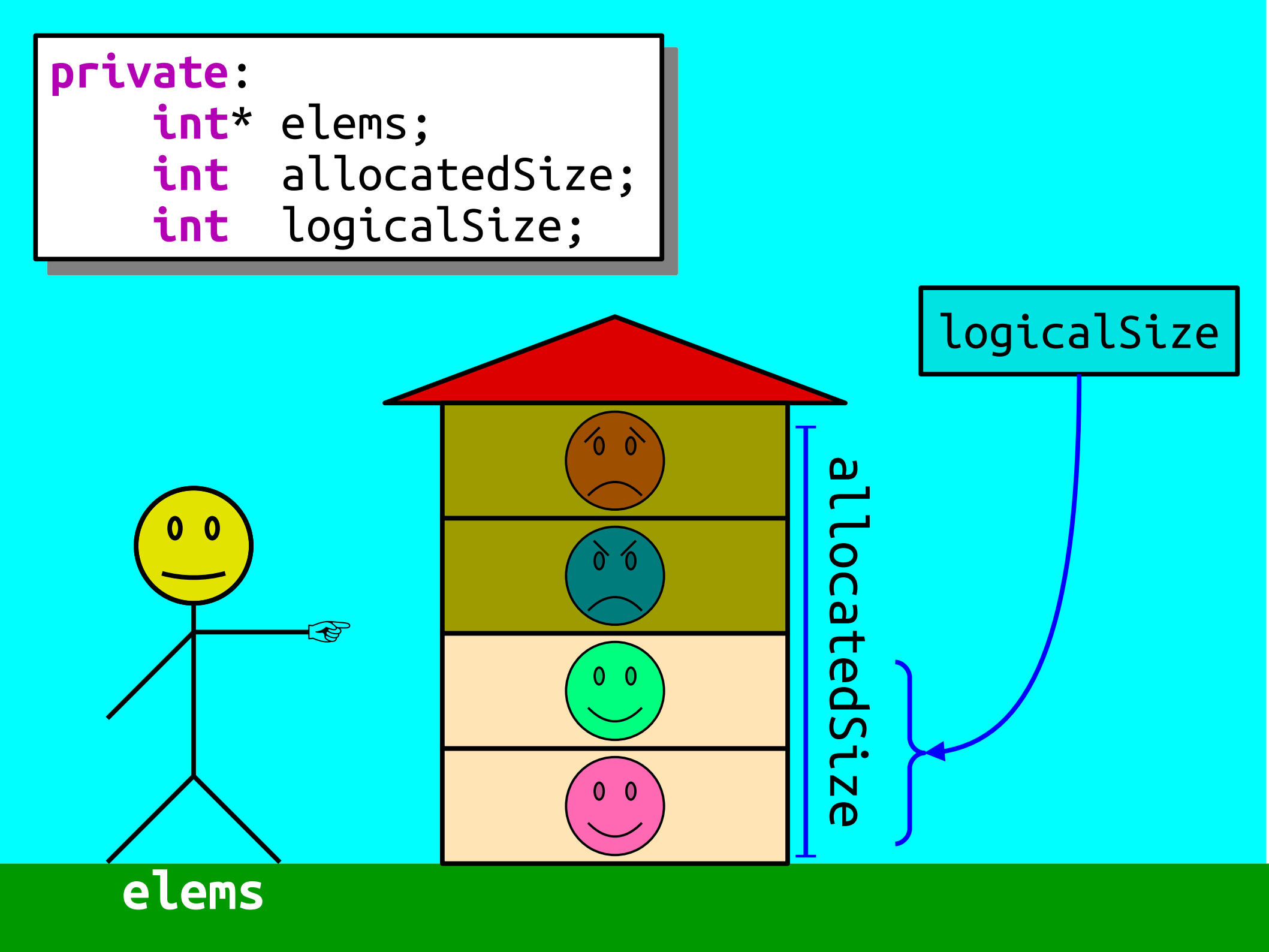
```
private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
```



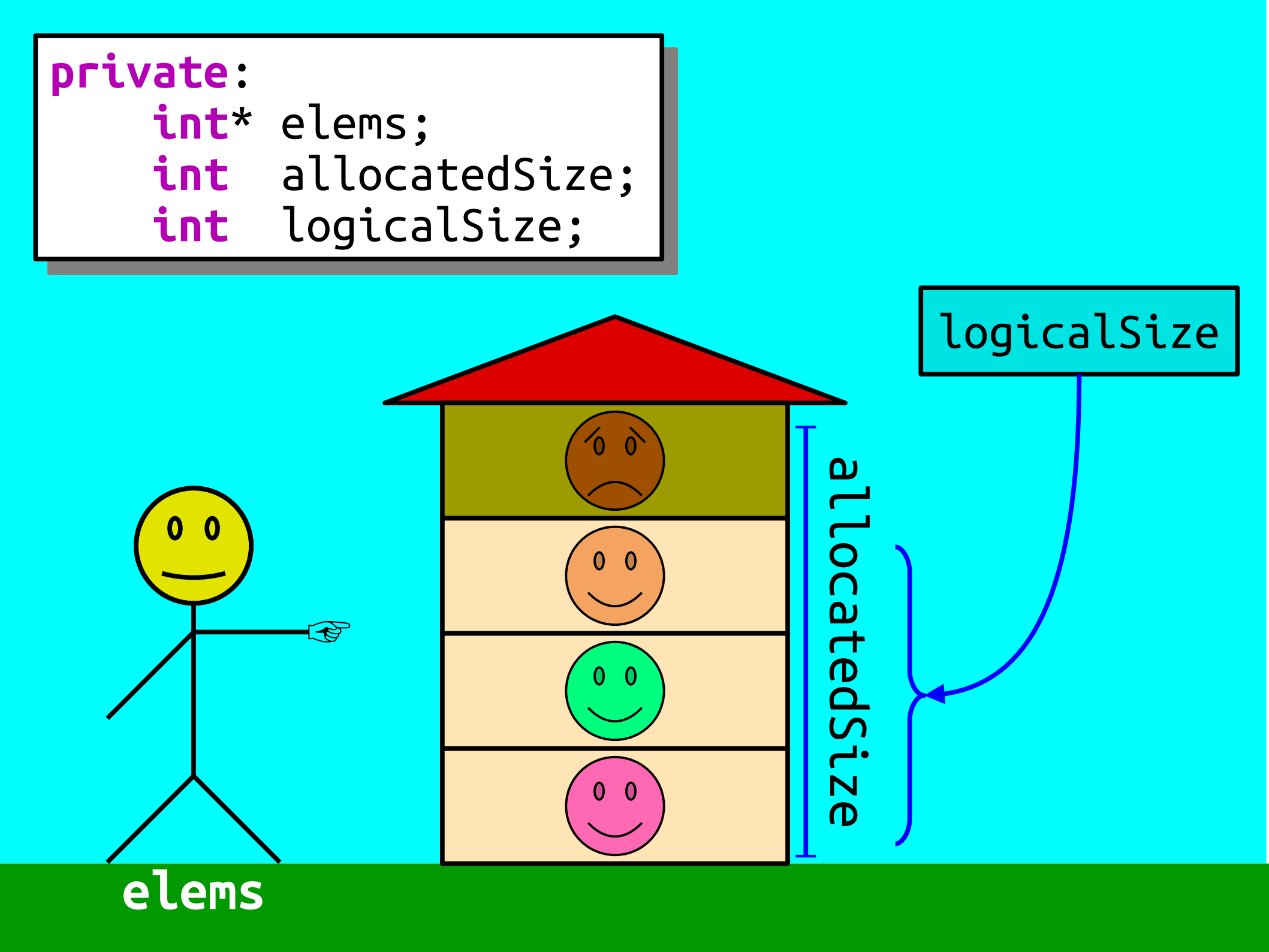
```
private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
```



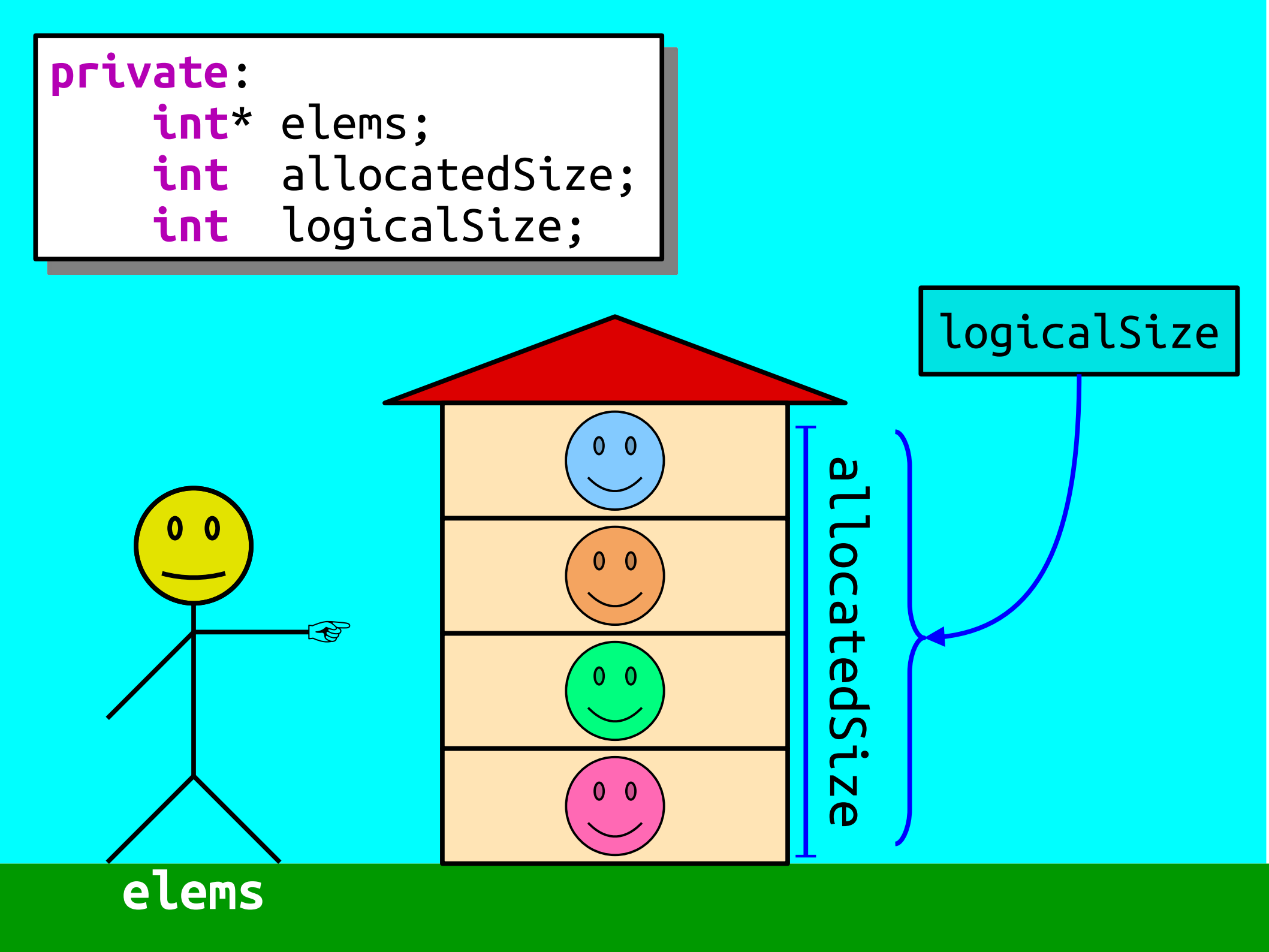
```
private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
```



```
private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
```



```
private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
```



```
class OurStack {
public:
    OurStack();

    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```

```
class OurStack {
public:
    OurStack();

    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```

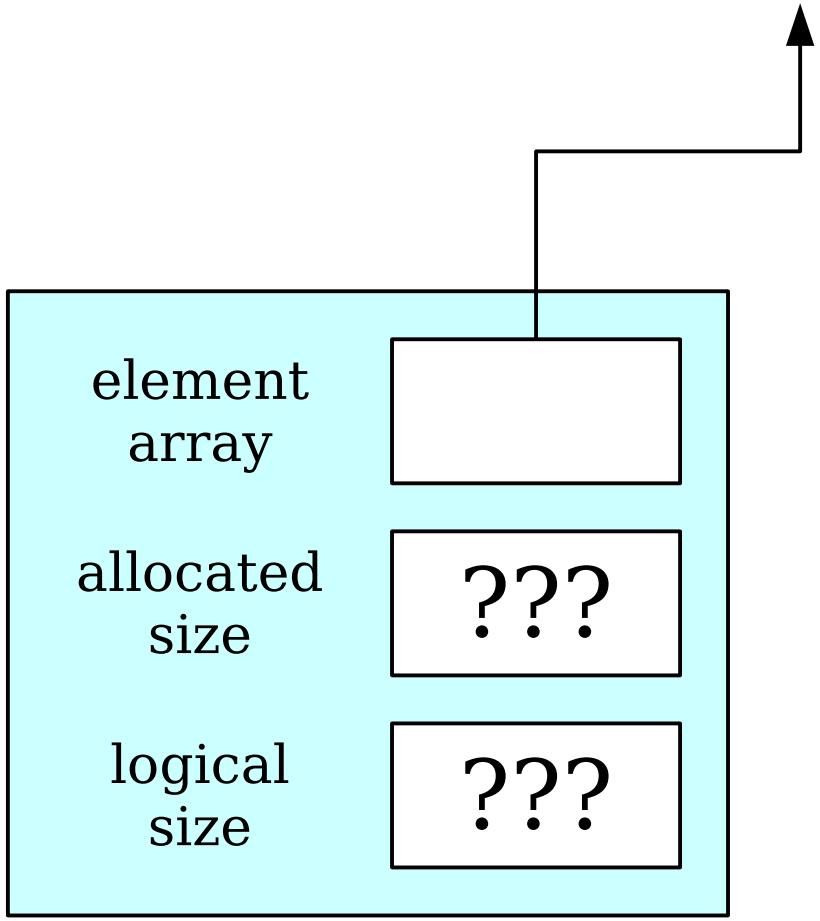
Cradle to Grave

```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave

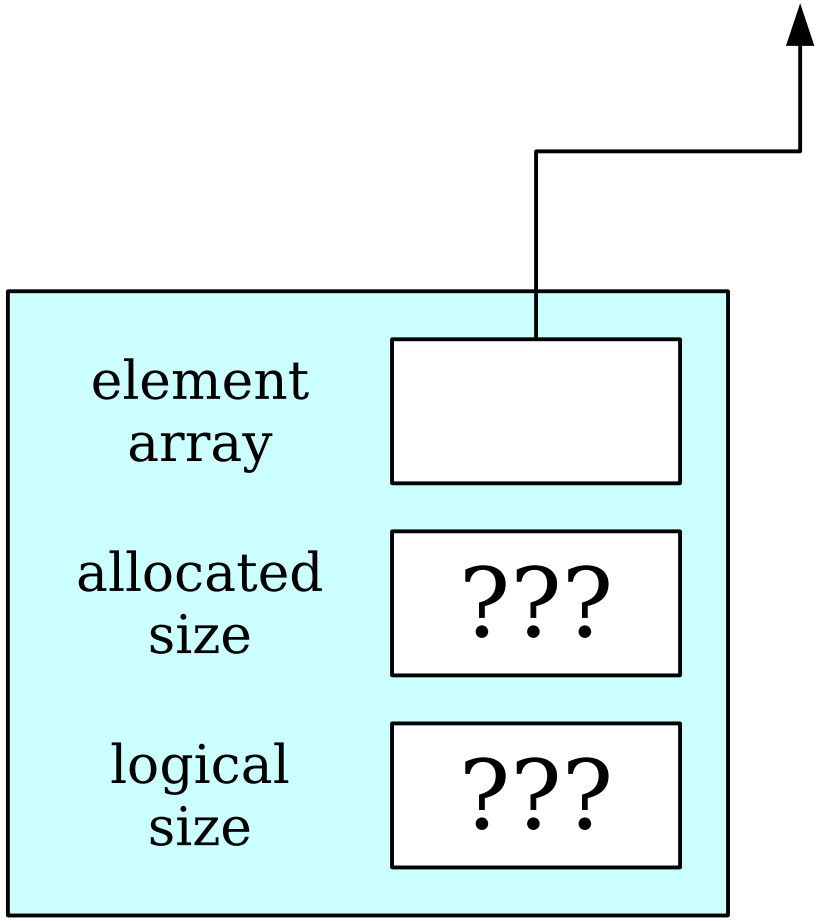
```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave




```
int main() {
   OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

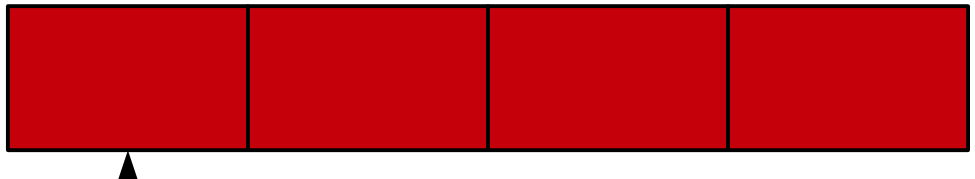
Cradle to Grave

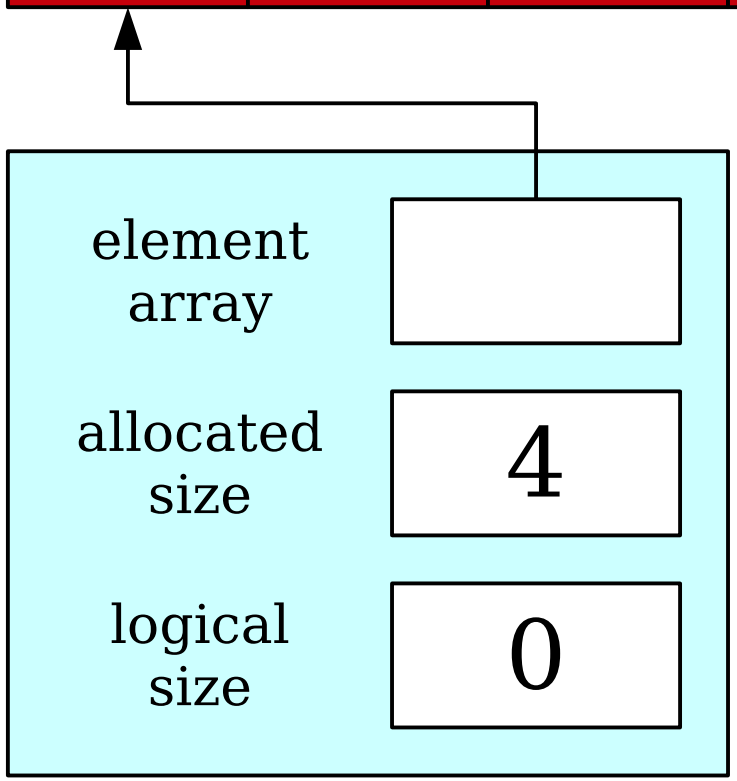



```
int
}
```

```
OurStack::OurStack() {
    logicalSize = 0;
    allocatedSize = kInitialSize;
    elems = new int[allocatedSize];
}
```

Cradle to Grave

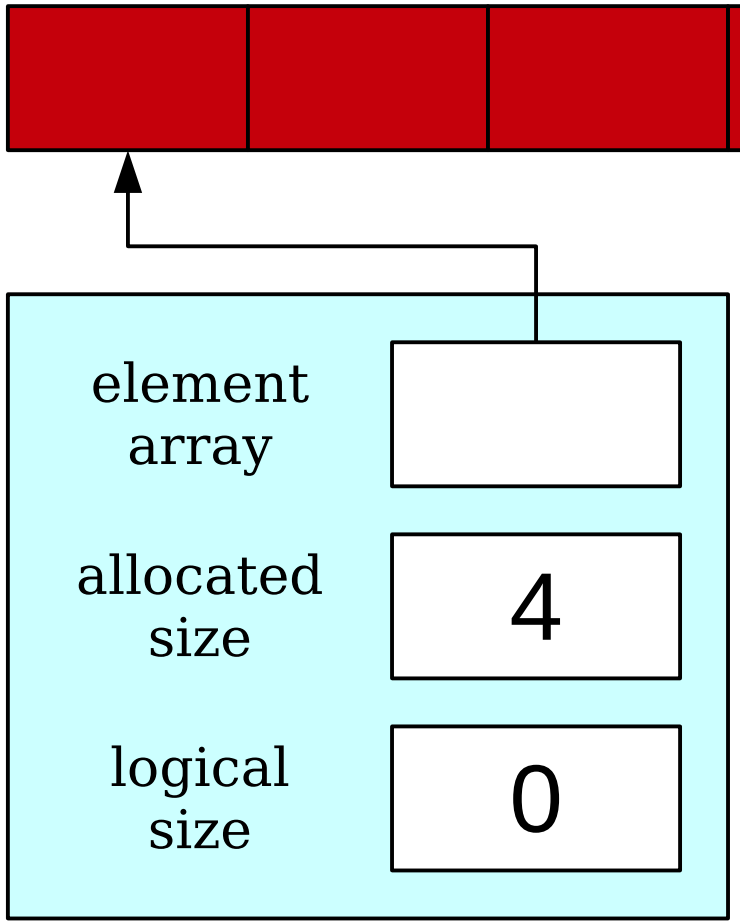





```
int

}
```

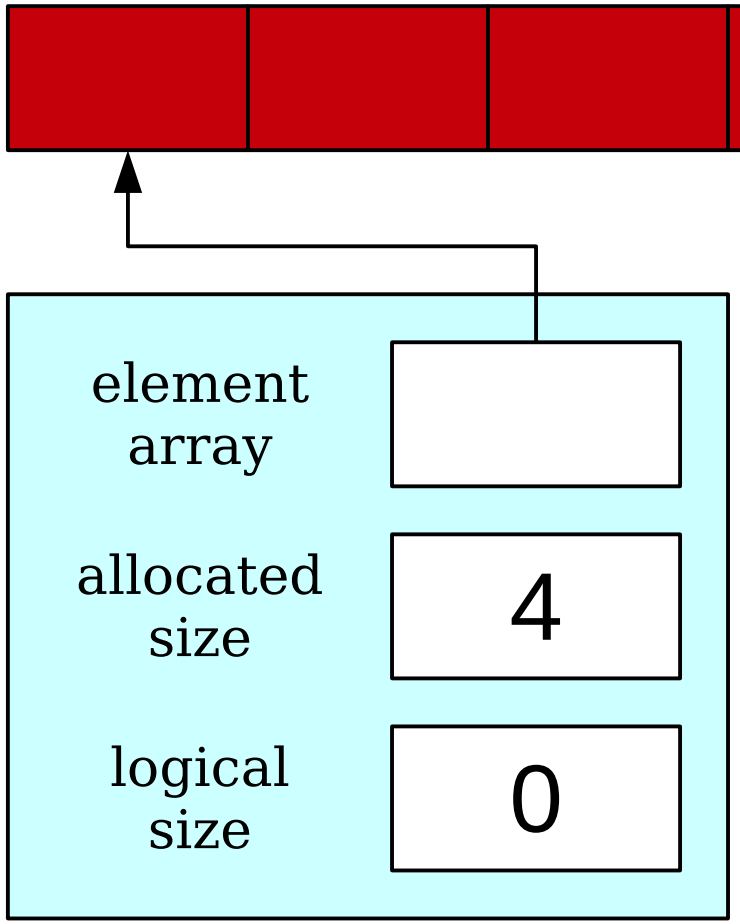
```
OurStack::OurStack() {
    logicalSize = 0;
    allocatedSize = kInitialSize;
    elems = new int[allocatedSize];
}
```

Cradle to Grave



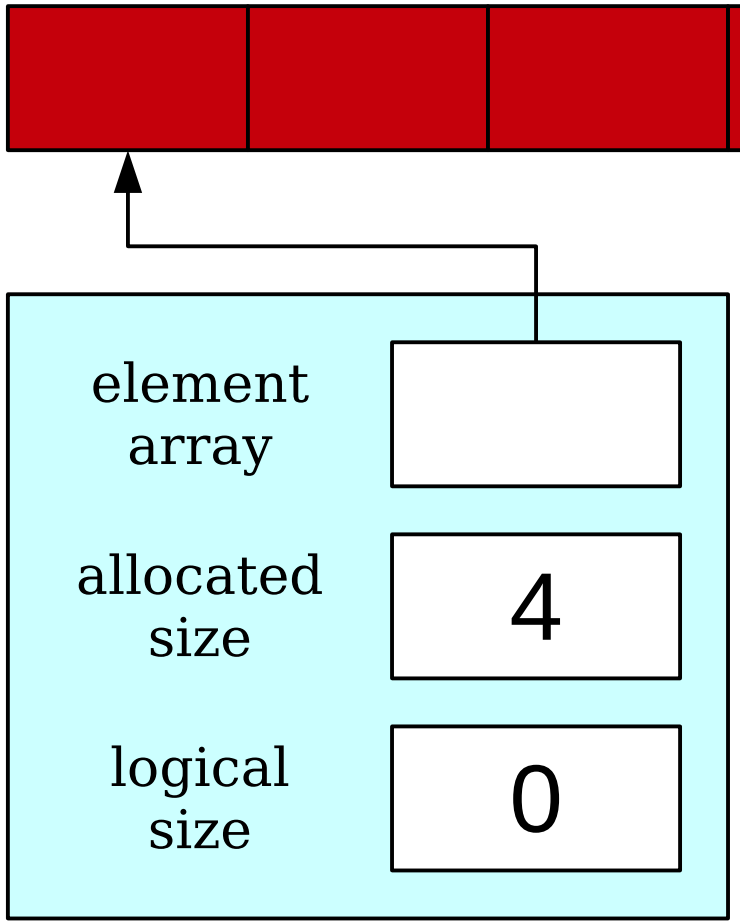

```
int main() {
   OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave



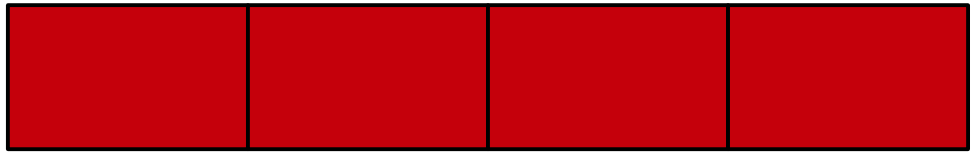

```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave



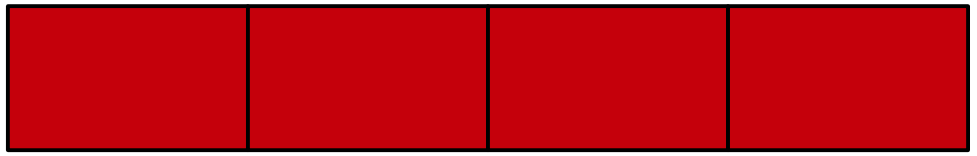

```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave



```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave




```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cleaning Up our Messes

Destructors

- A ***destructor*** is a special member function responsible for cleaning up an object's memory.
- It's automatically called whenever an object's lifetime ends (for example, if it's a local variable that goes out of scope.)
- The destructor for a class named ***ClassName*** has signature

  ~***ClassName***();

```
class OurStack {
public:
    OurStack();
    ~OurStack();

    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```

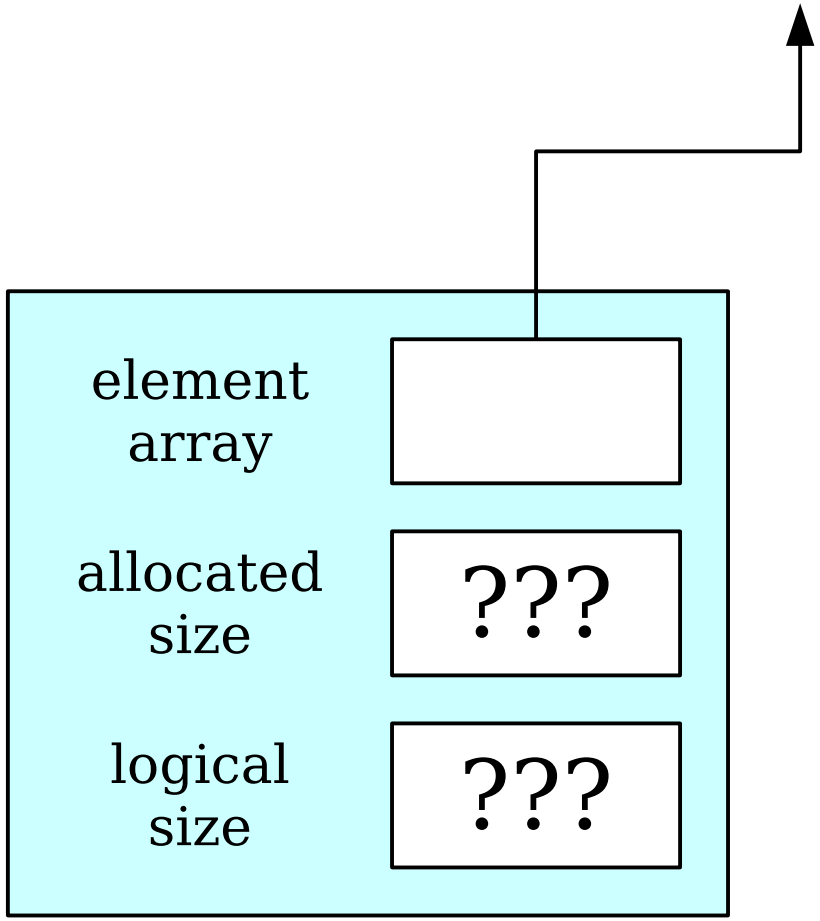
Cradle to Grave

```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave

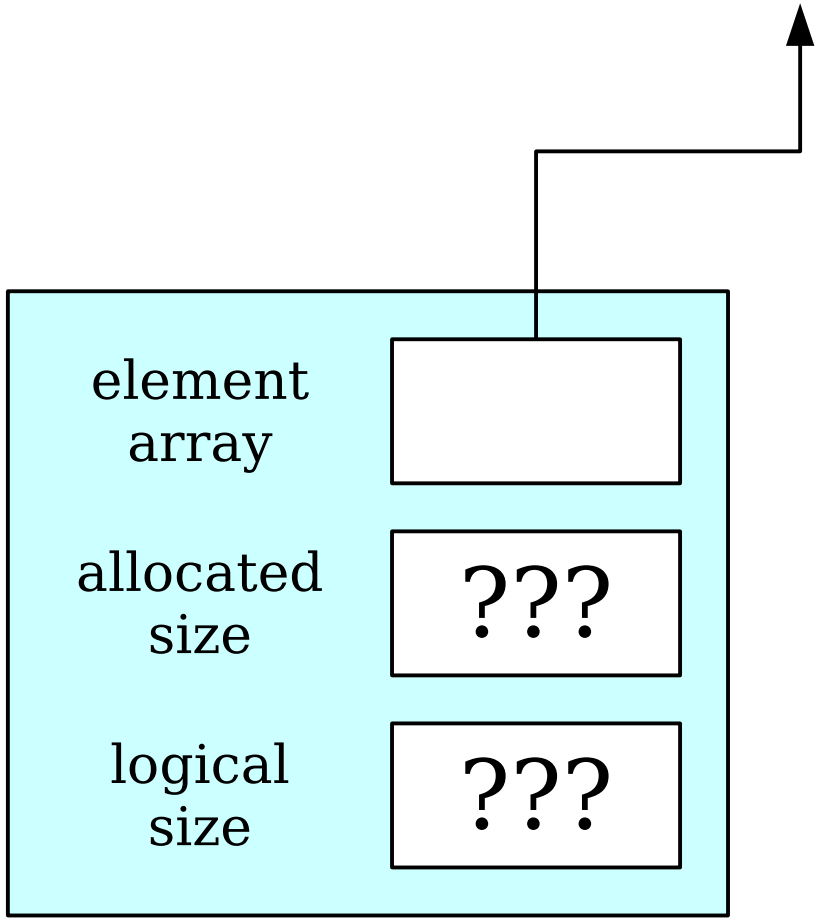
```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave




```
int main() {
   OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

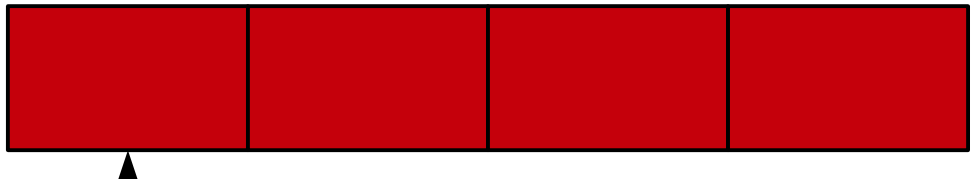
Cradle to Grave

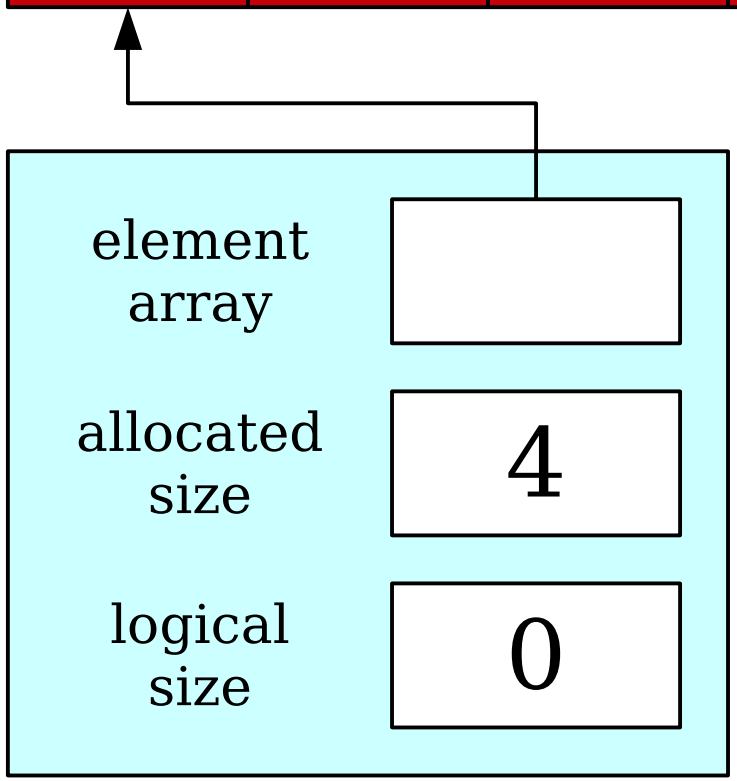



```
int
}
```

```
OurStack::OurStack() {
    logicalSize = 0;
    allocatedSize = kInitialSize;
    elems = new int[allocatedSize];
}
```

Cradle to Grave

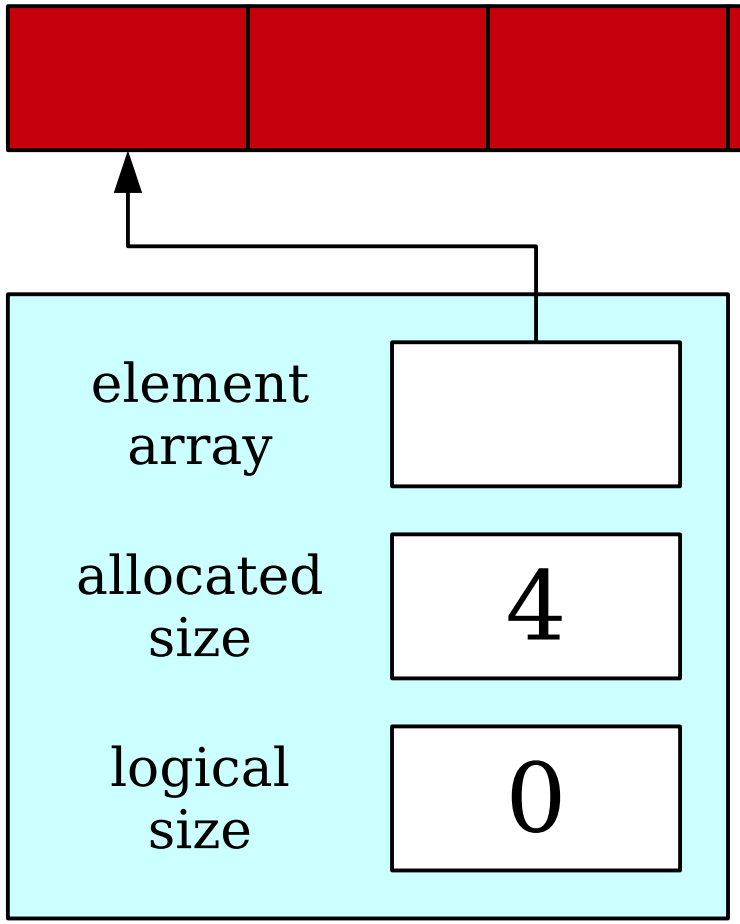





```
int

}
```

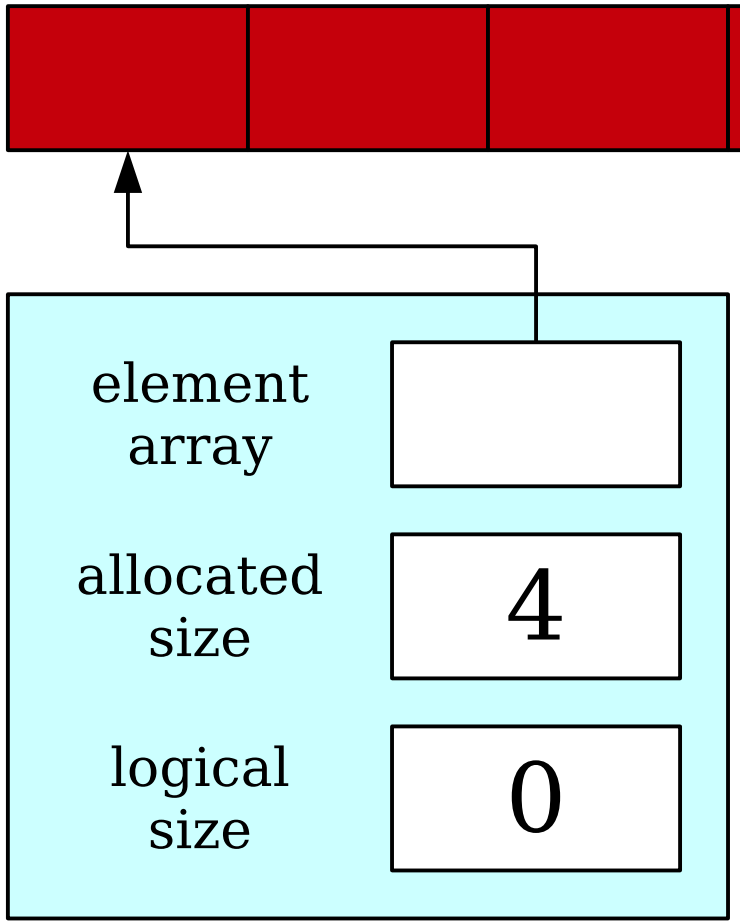
```
OurStack::OurStack() {
    logicalSize = 0;
    allocatedSize = kInitialSize;
    elems = new int[allocatedSize];
}
```

Cradle to Grave



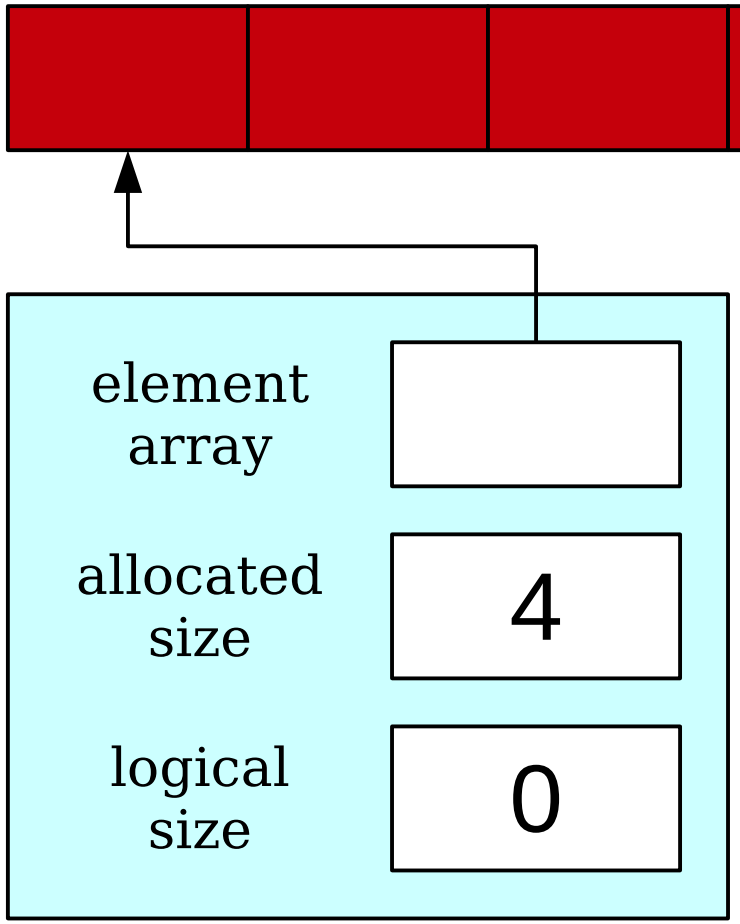

```
int main() {
   OurStack stack;

   /* The stack lives a rich, happy,
    * fulfilling life, the kind we
    * all aspire to.
    */

   return 0;
}
```

Cradle to Grave



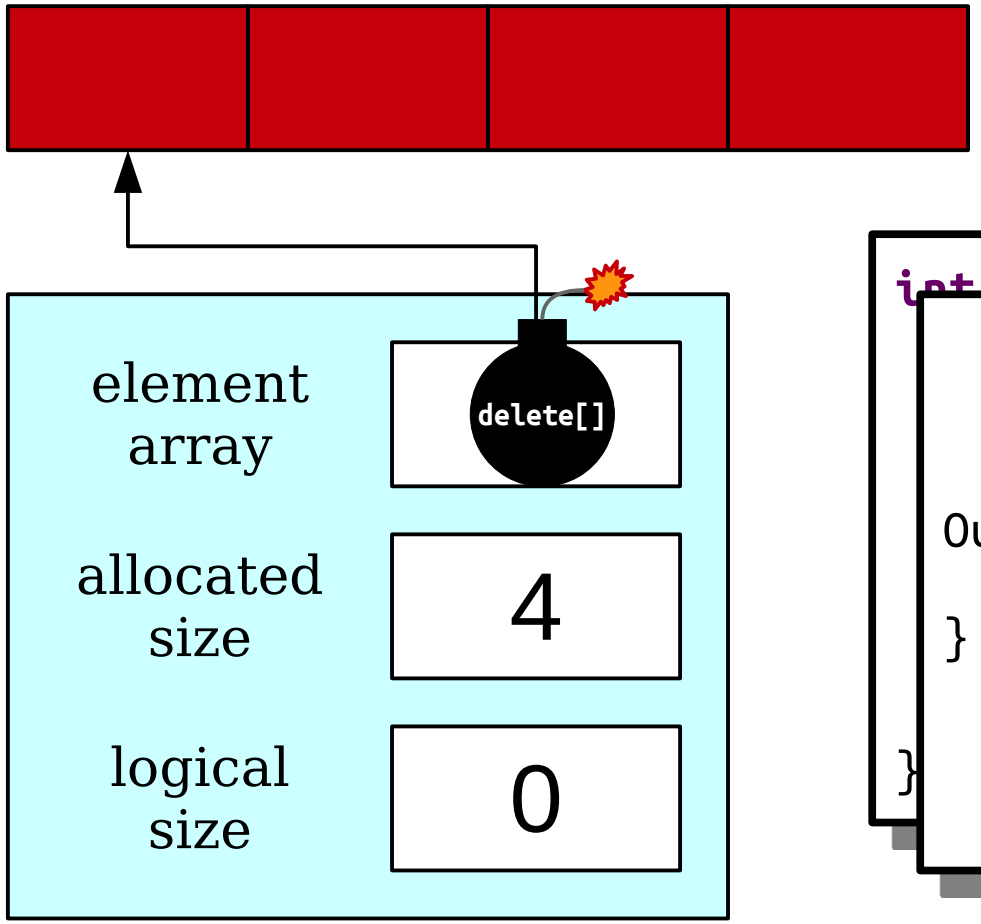

```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave



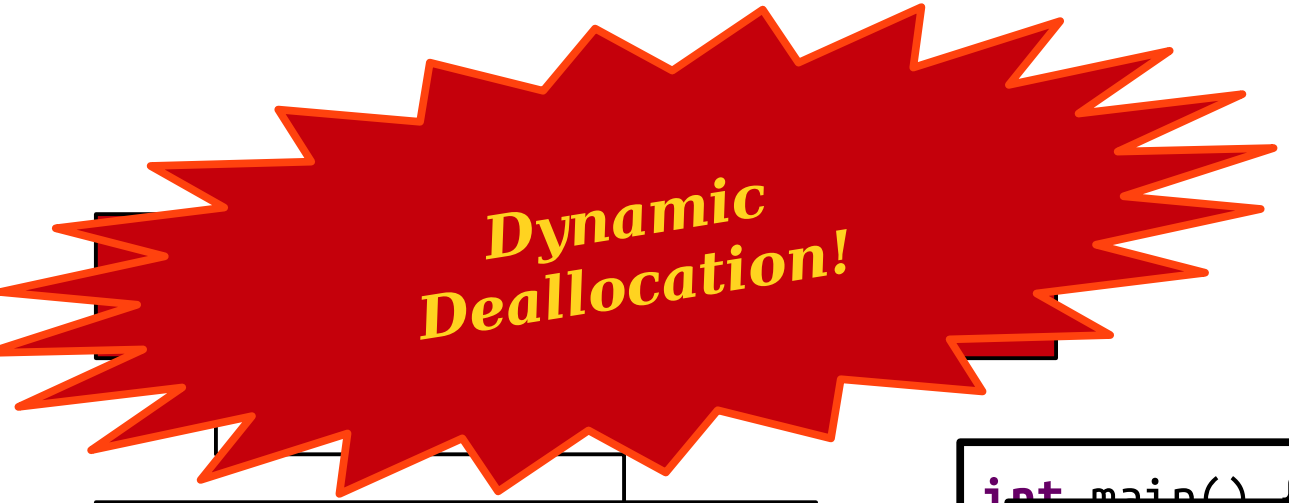

```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

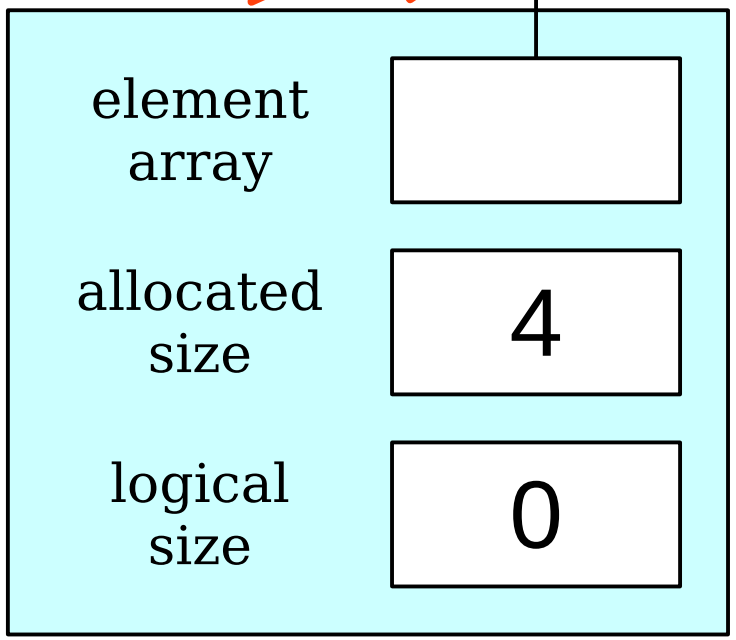
Cradle to Grave




```
int main() {


OurStack::~OurStack() {
    delete[] elems;
}


}
```

Cradle to Grave

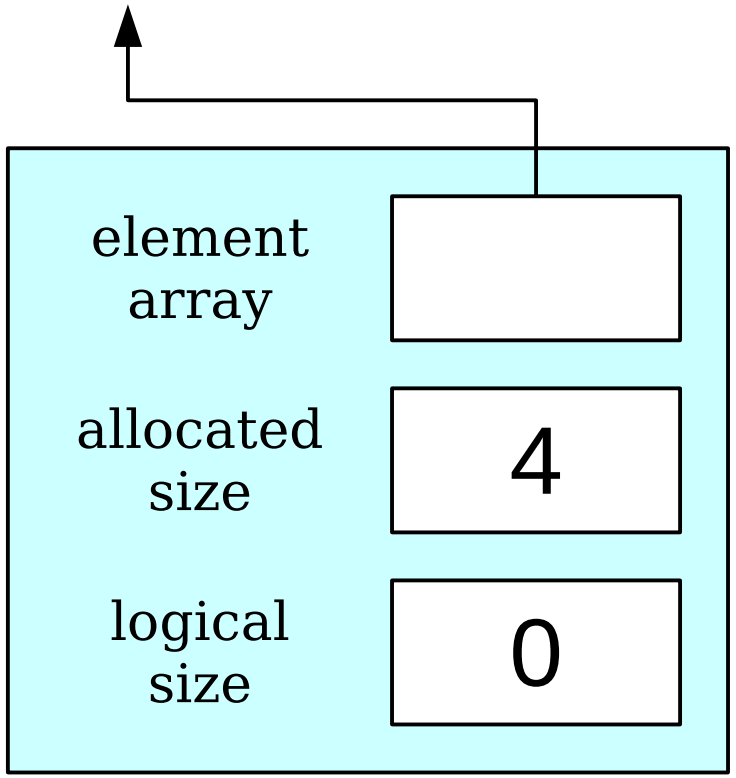





```
int main() {


}
```

```
OurStack::~OurStack() {
    delete[] elems;
}
```

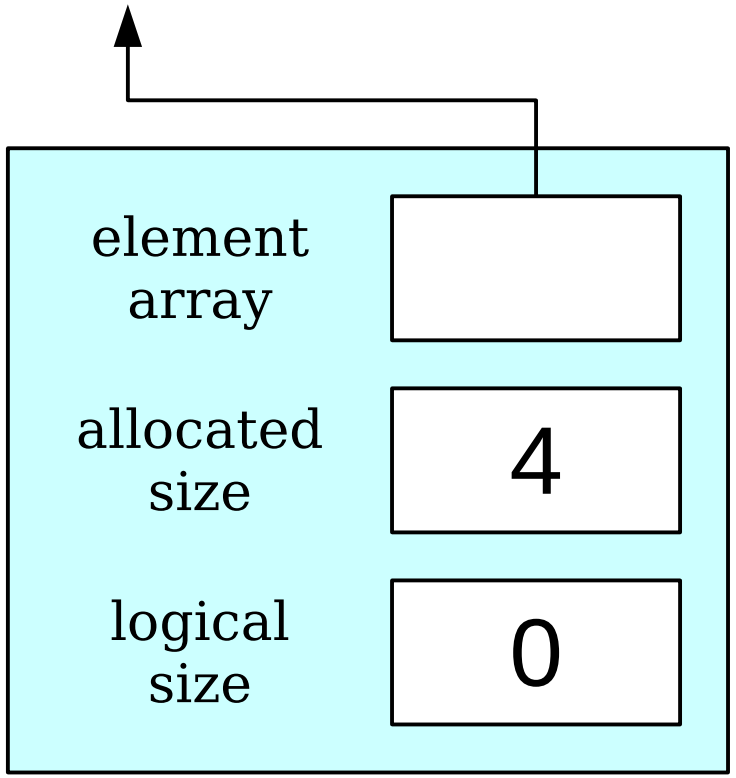
Cradle to Grave




```
int main() {


}
```

```
OurStack::~OurStack() {
    delete[] elems;
}
```

Cradle to Grave

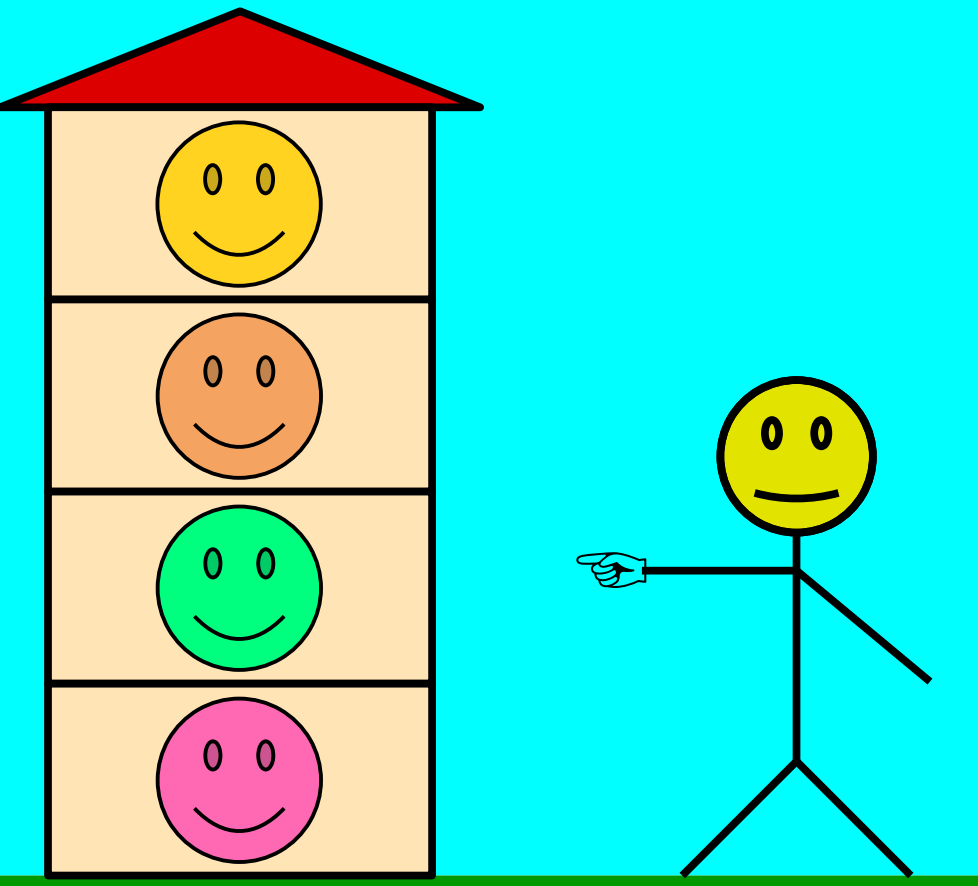



```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave

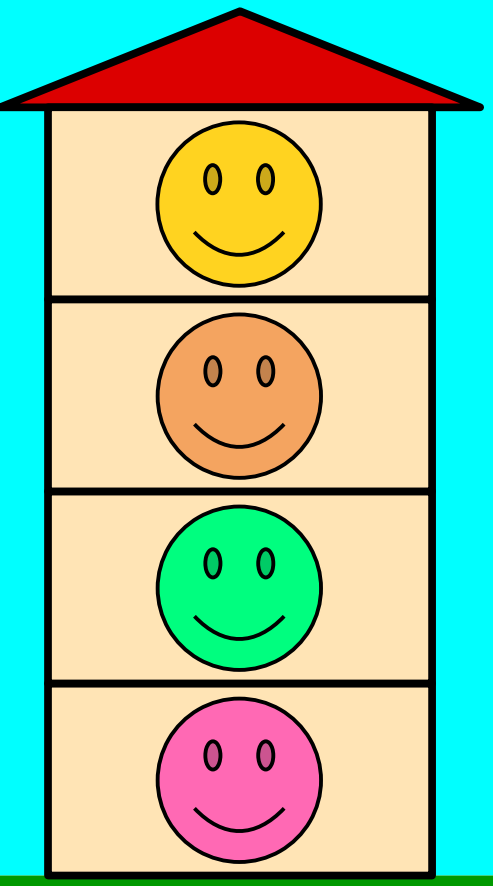
```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

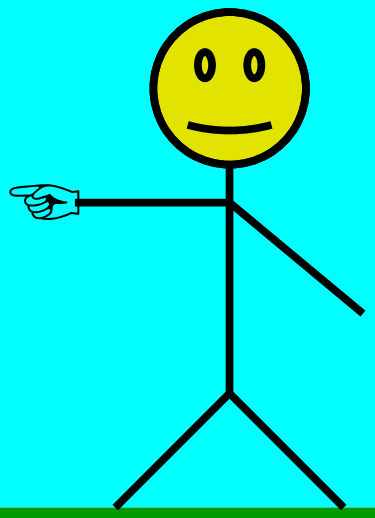
Getting More Space



elems

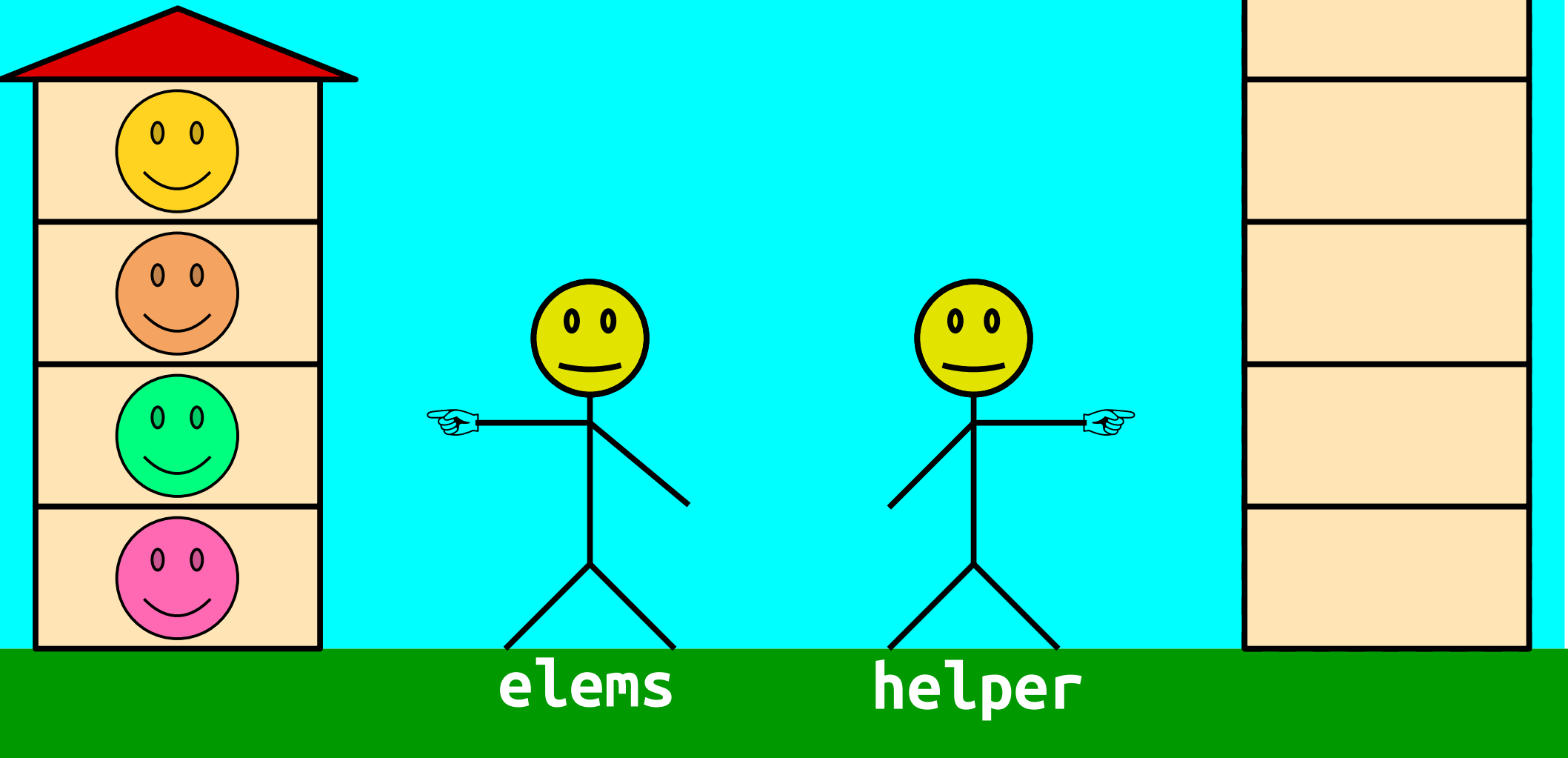
```
allocatedSize = /* bigger */;
```



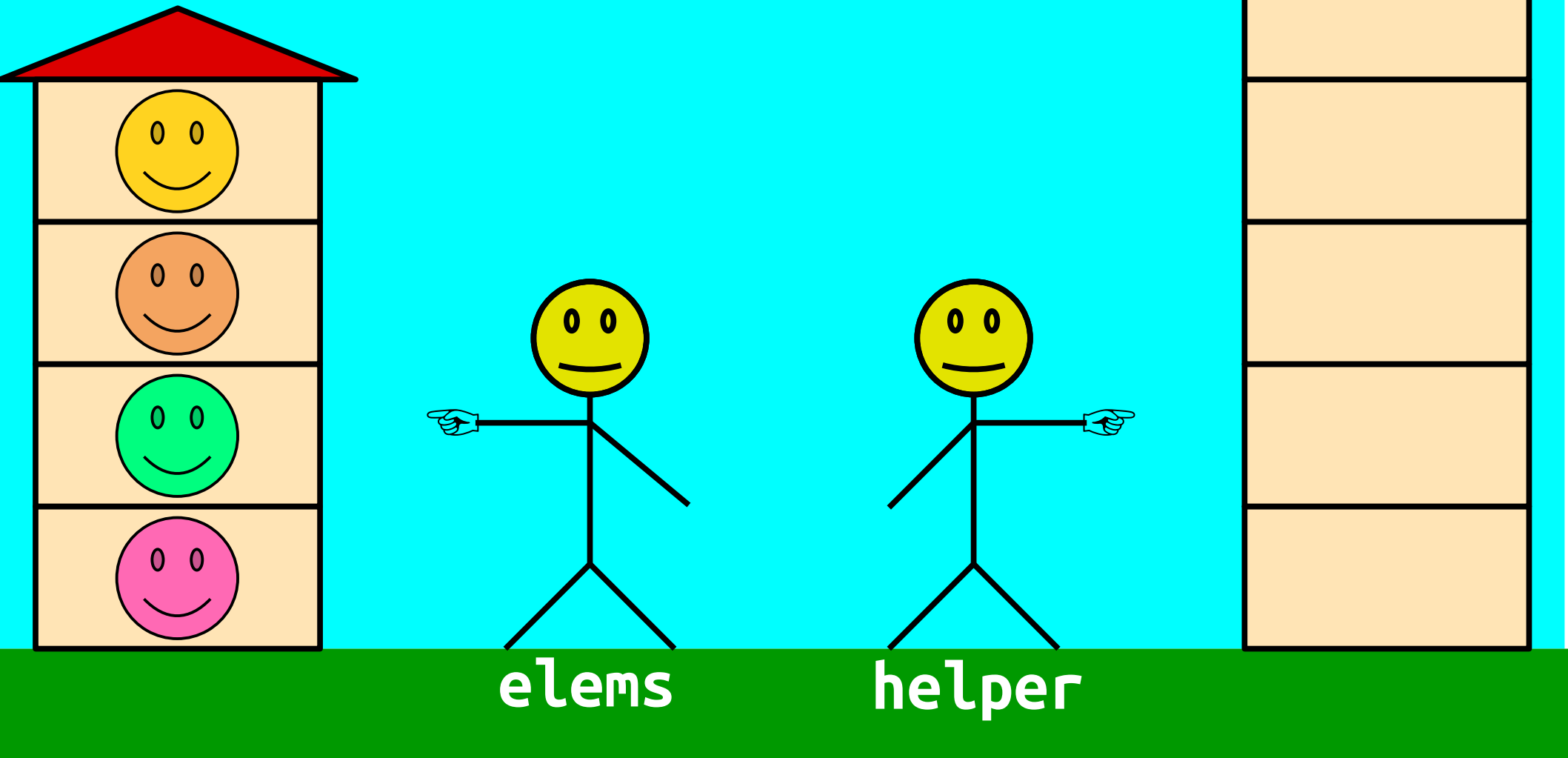


elems

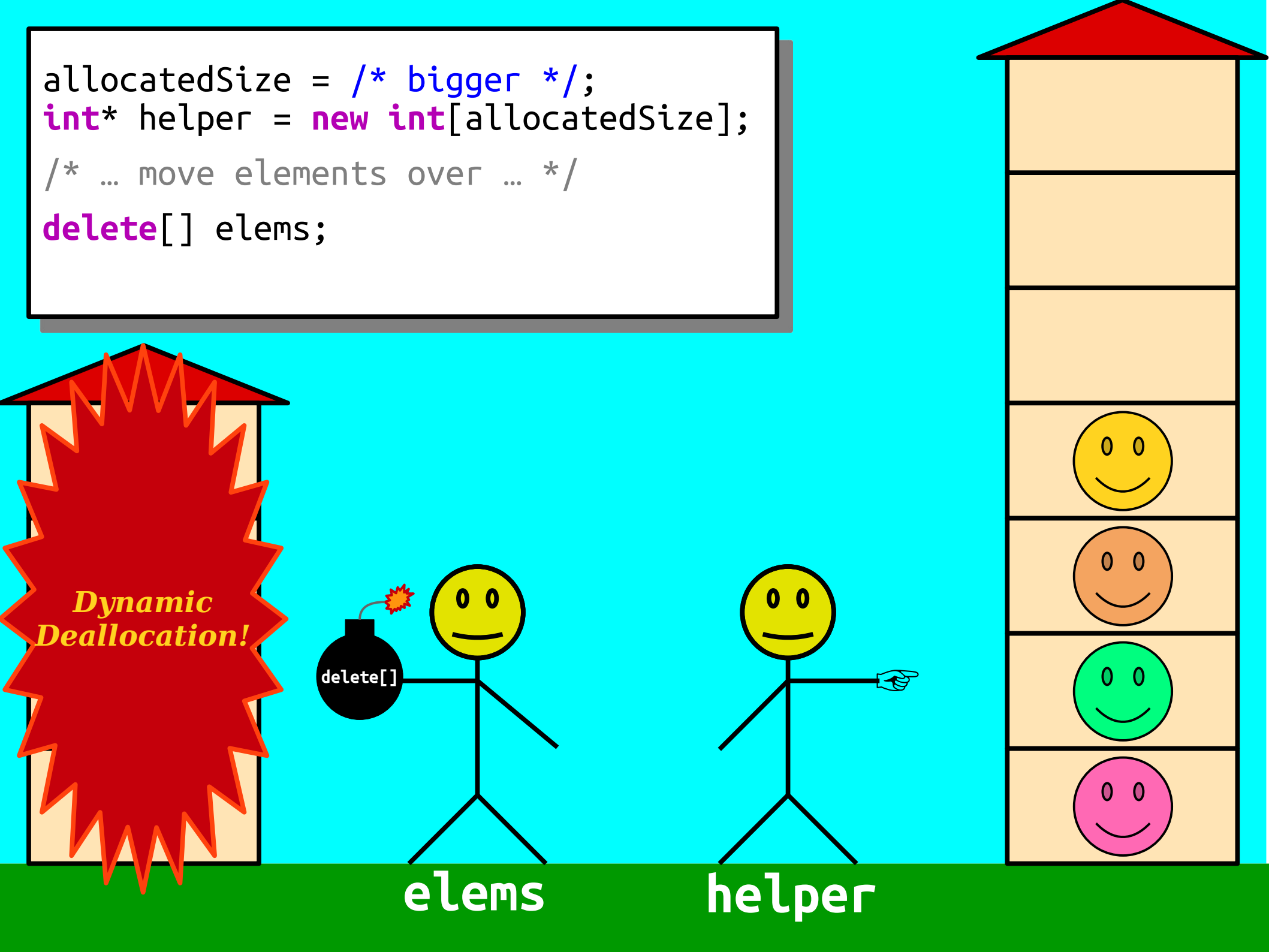
```
allocatedSize = /* bigger */;
int* helper = new int[allocatedSize];
```



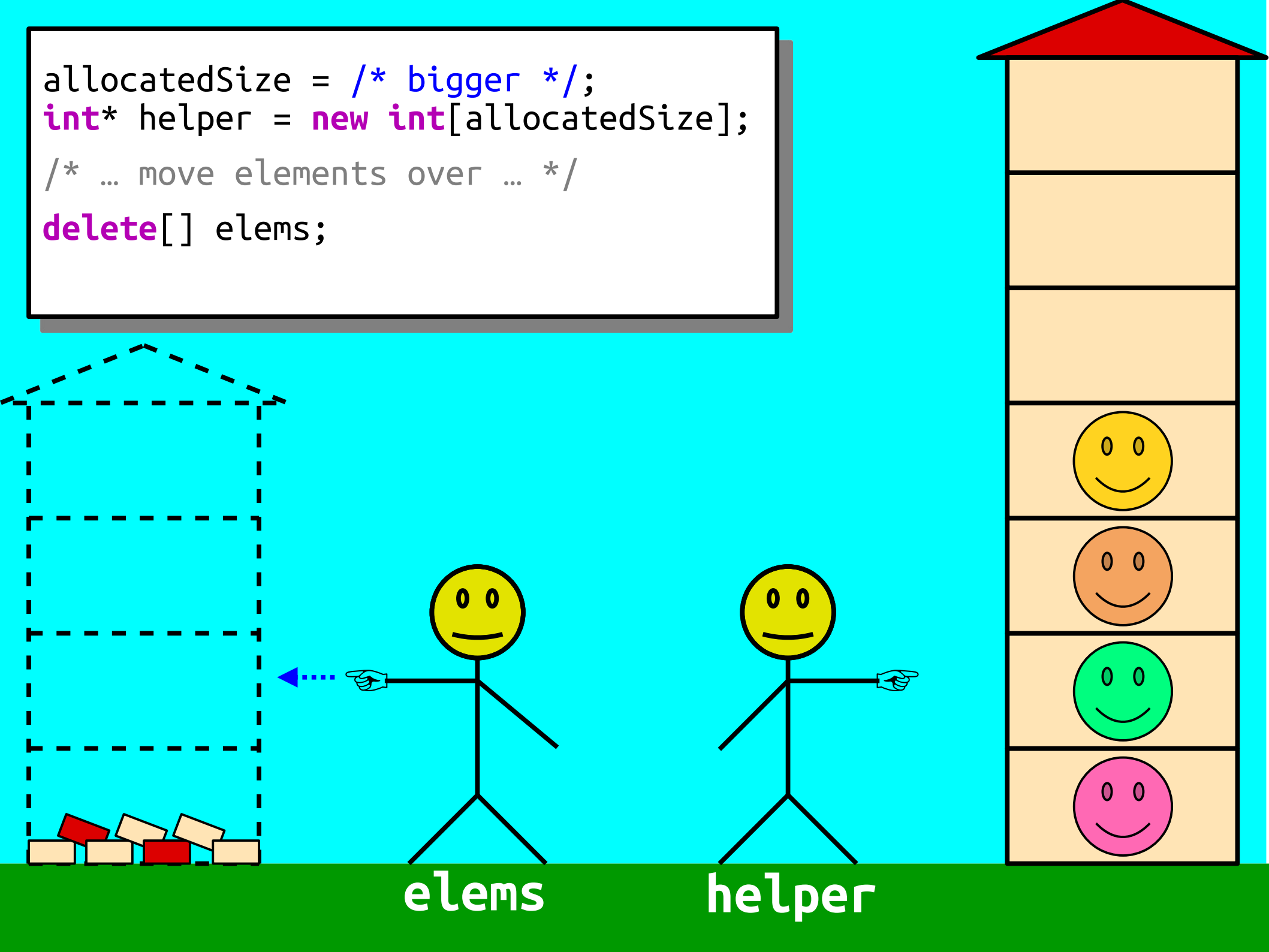
```
allocatedSize = /* bigger */;
int* helper = new int[allocatedSize];
/* … move elements over … */
```



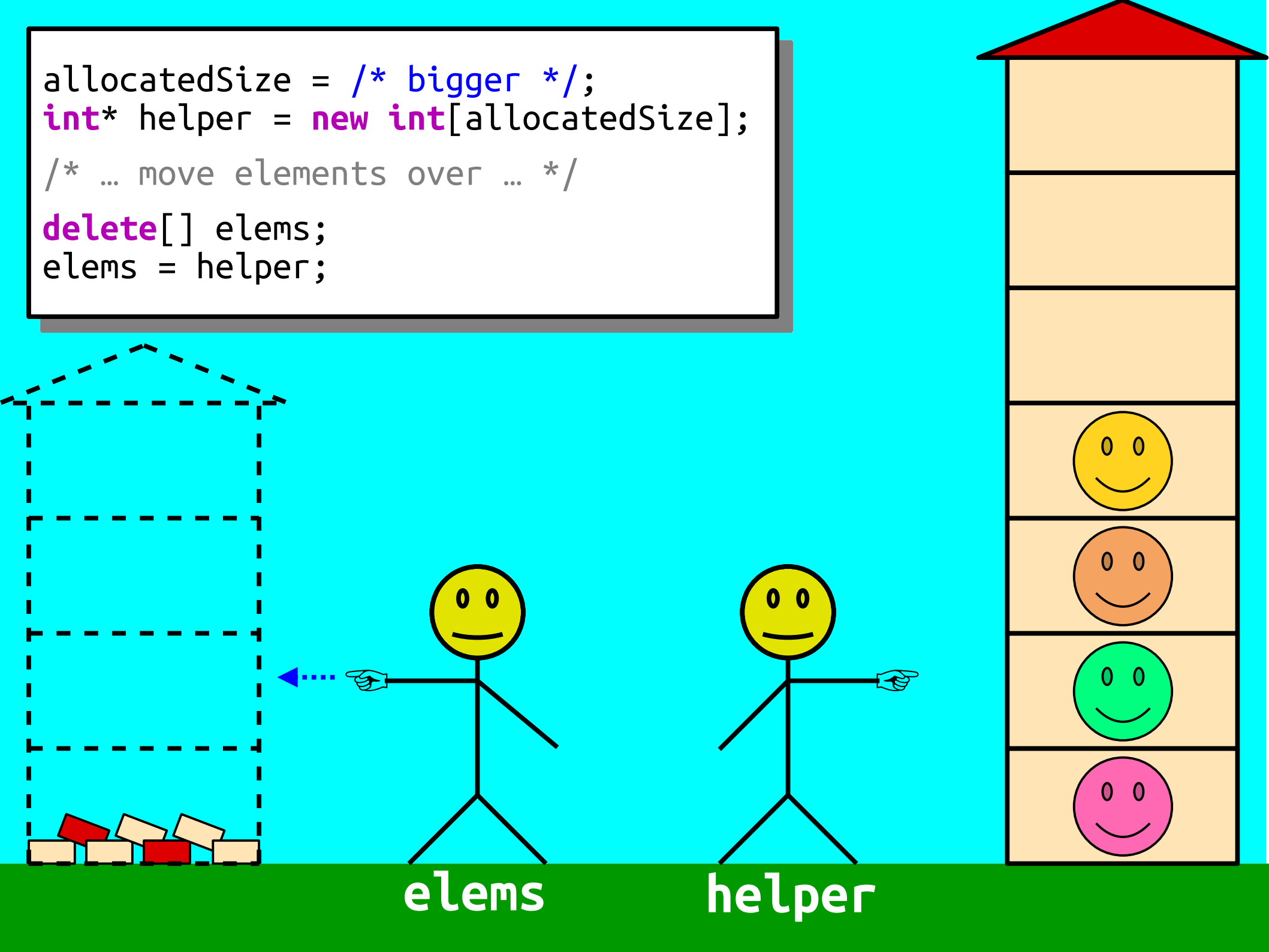
```
allocatedSize = /* bigger */;
int* helper = new int[allocatedSize];
/* … move elements over … */
delete[] elems;
```



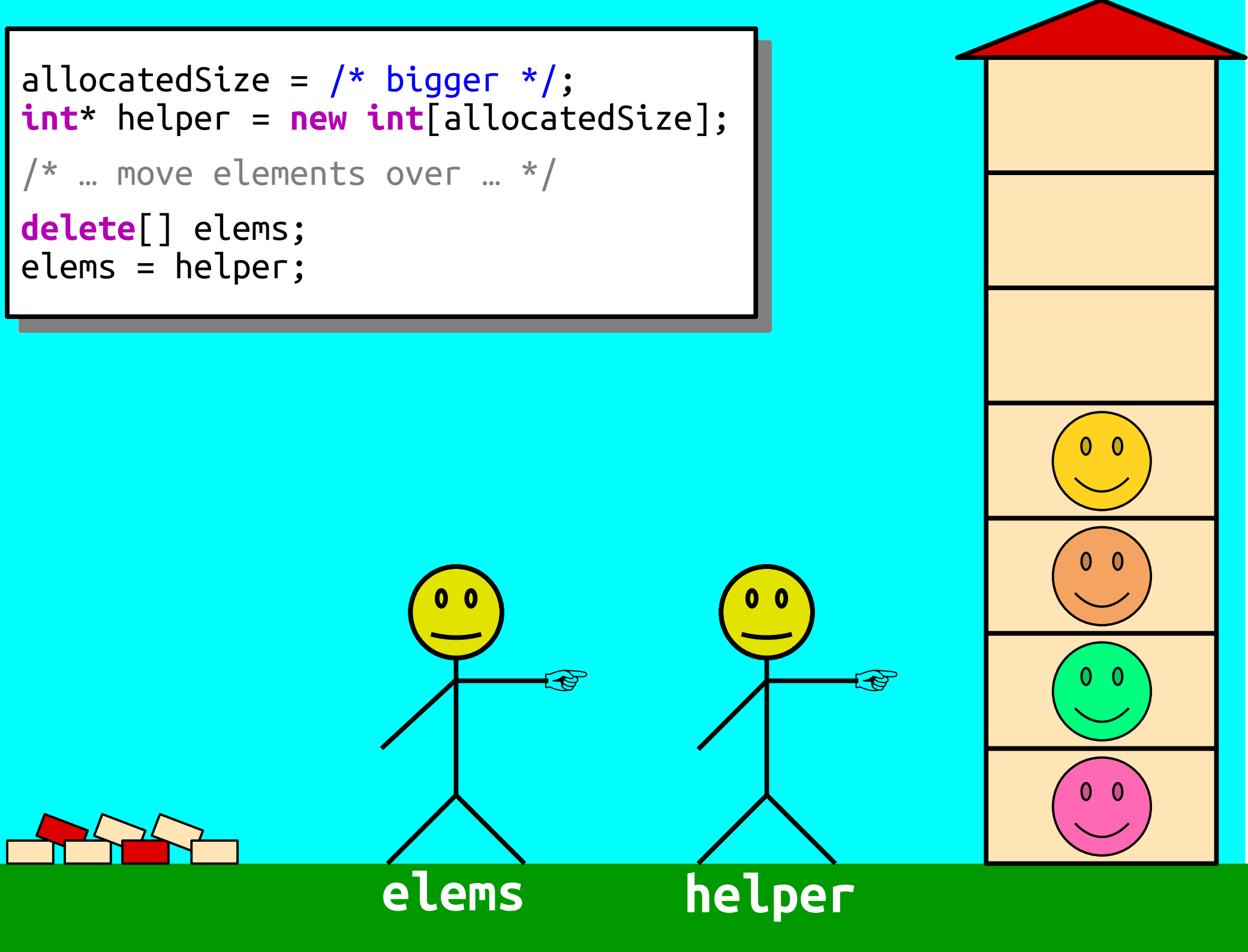
```
allocatedSize = /* bigger */;
int* helper = new int[allocatedSize];
/* … move elements over … */
delete[] elems;
```



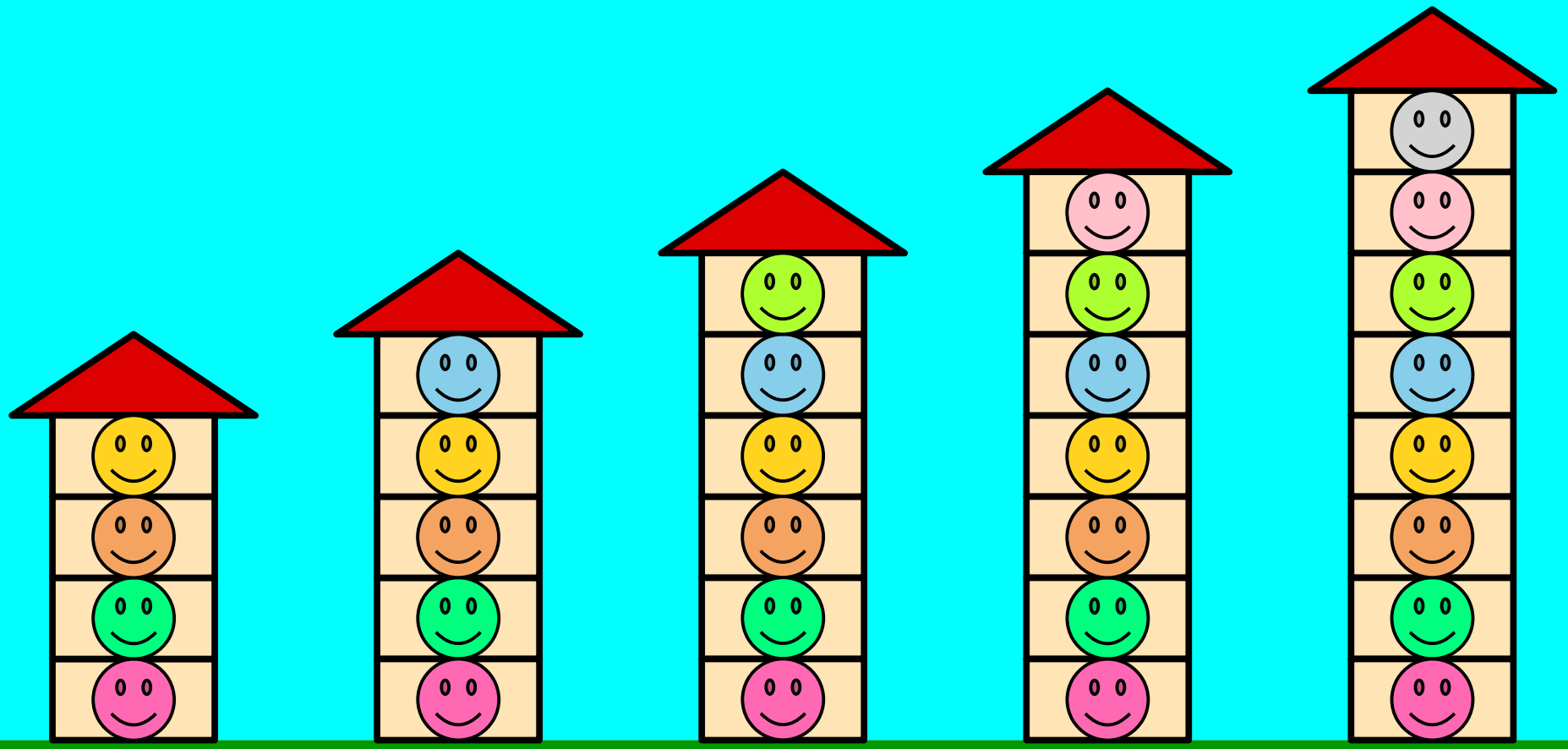
```
allocatedSize = /* bigger */;
int* helper = new int[allocatedSize];
/* … move elements over … */
delete[] elems;
elems = helper;
```



```
allocatedSize = /* bigger */;
int* helper = new int[allocatedSize];
/* … move elements over … */
delete[] elems;
elems = helper;
```

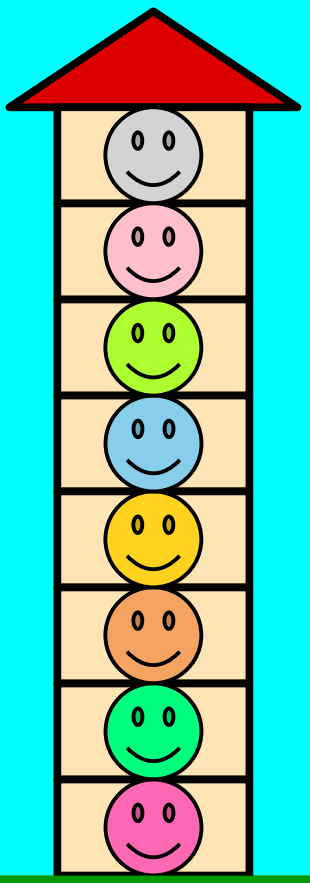


What is the big-O cost of a push?
What is the big-O cost of n pushes?



Every push beyond the first few requires moving all n elements from the old array to the new array.

Cost of a single push: **O(*n*)**.



4 Items Moved

5 Items Moved

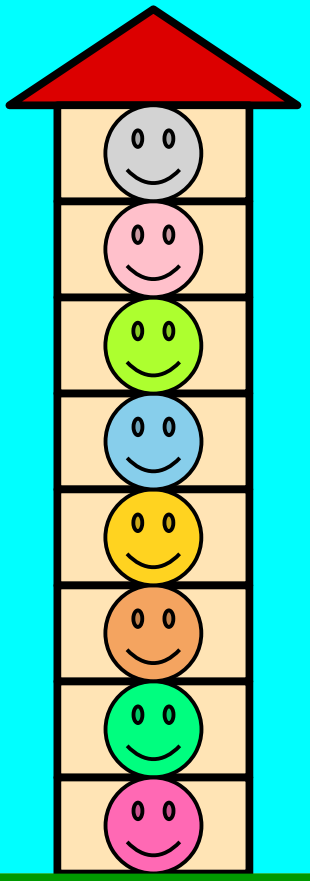
6 Items Moved

7 Items Moved

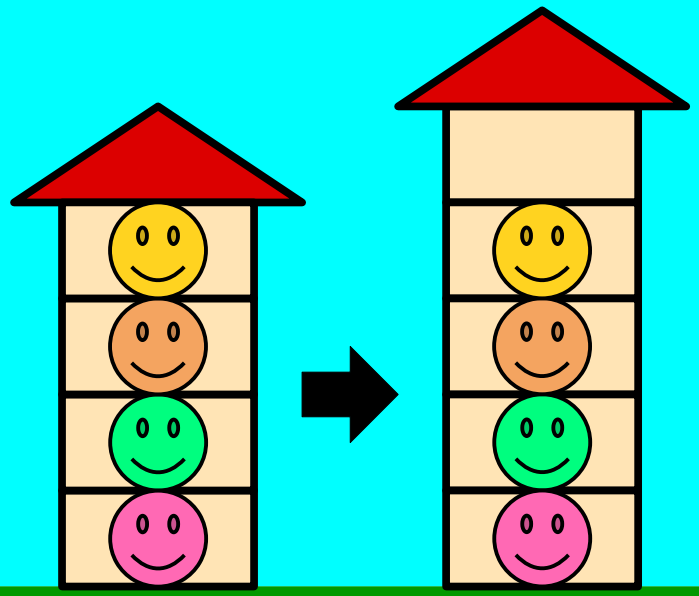
Every push beyond the first few requires moving all n elements from the old array to the new array.

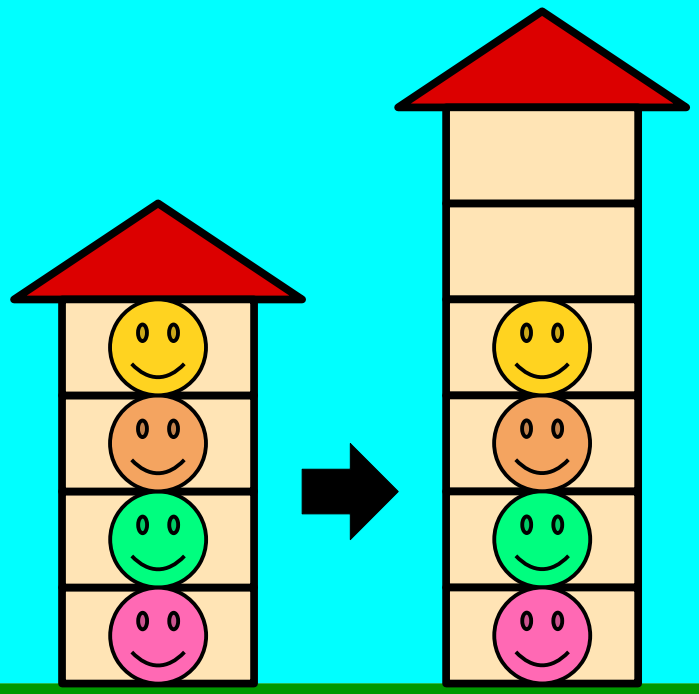
Cost of doing n pushes:
$4 + 5 + 6 + \ldots + n = \mathbf{O(n^2)}$.

Question: How do we speed this up?



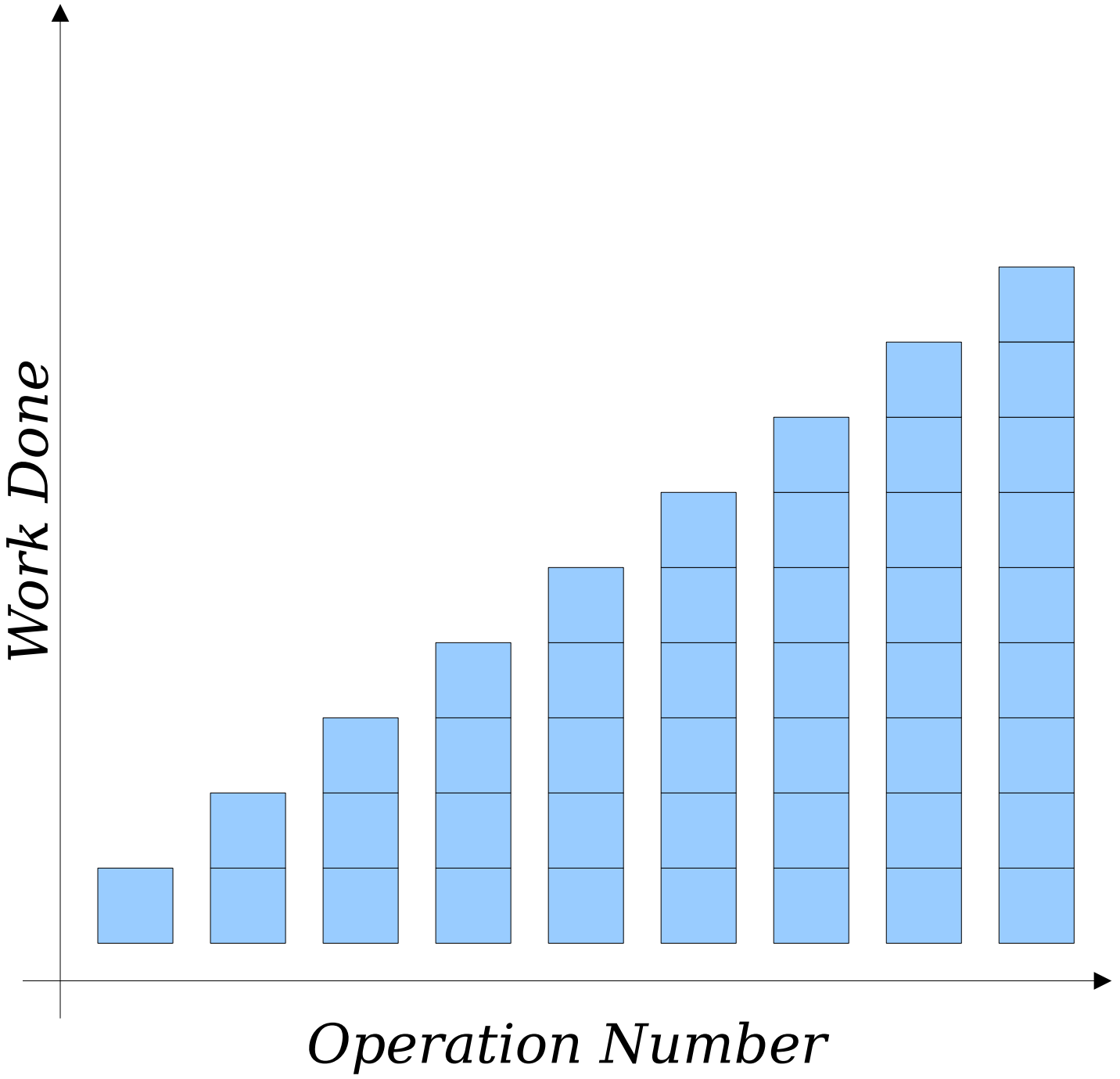
4 Items Moved **5 Items Moved** **6 Items Moved** **7 Items Moved**



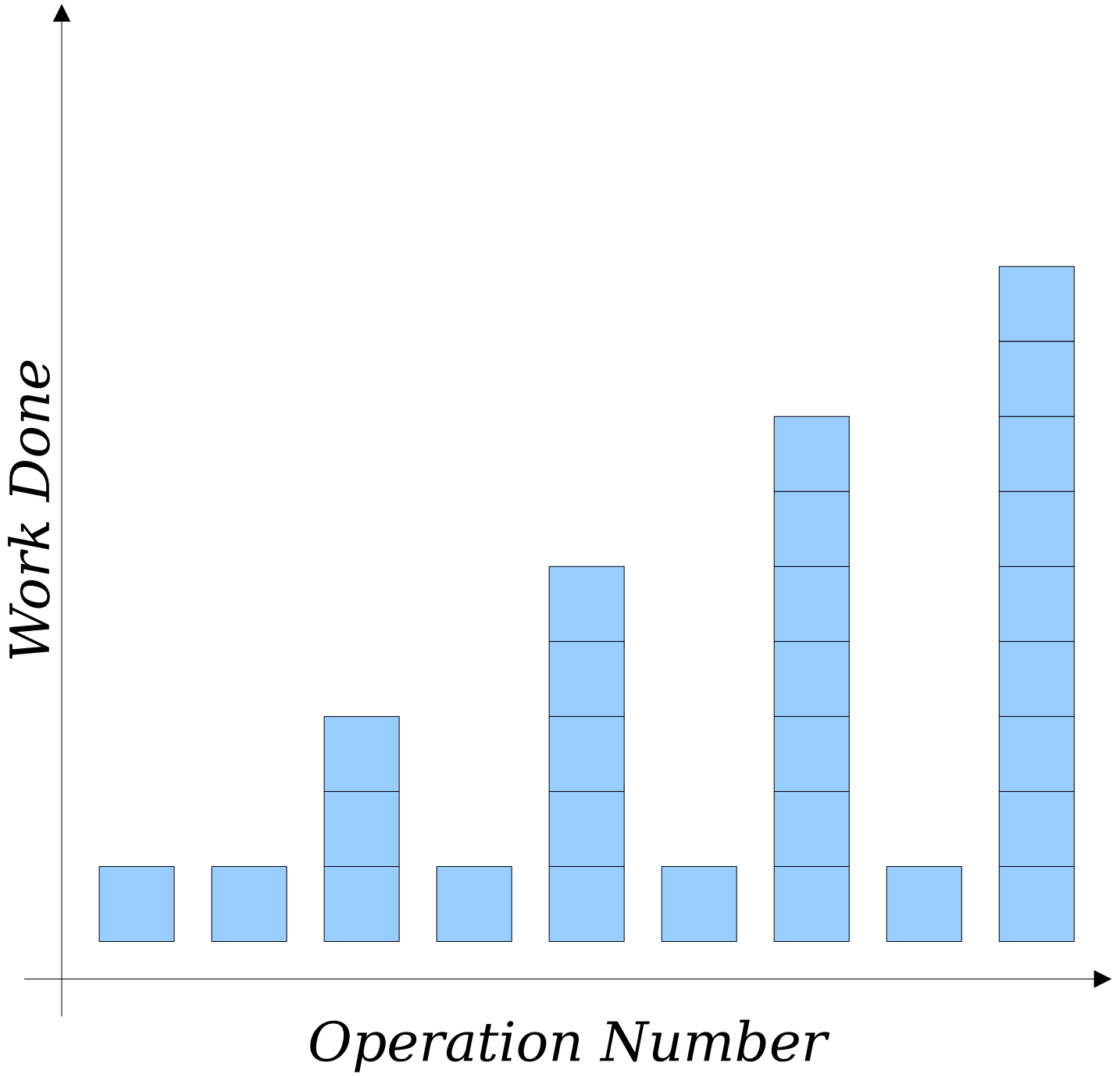


Now, only half the pushes we do will require moving everything to a new array.

Increase array size by ***adding one***.



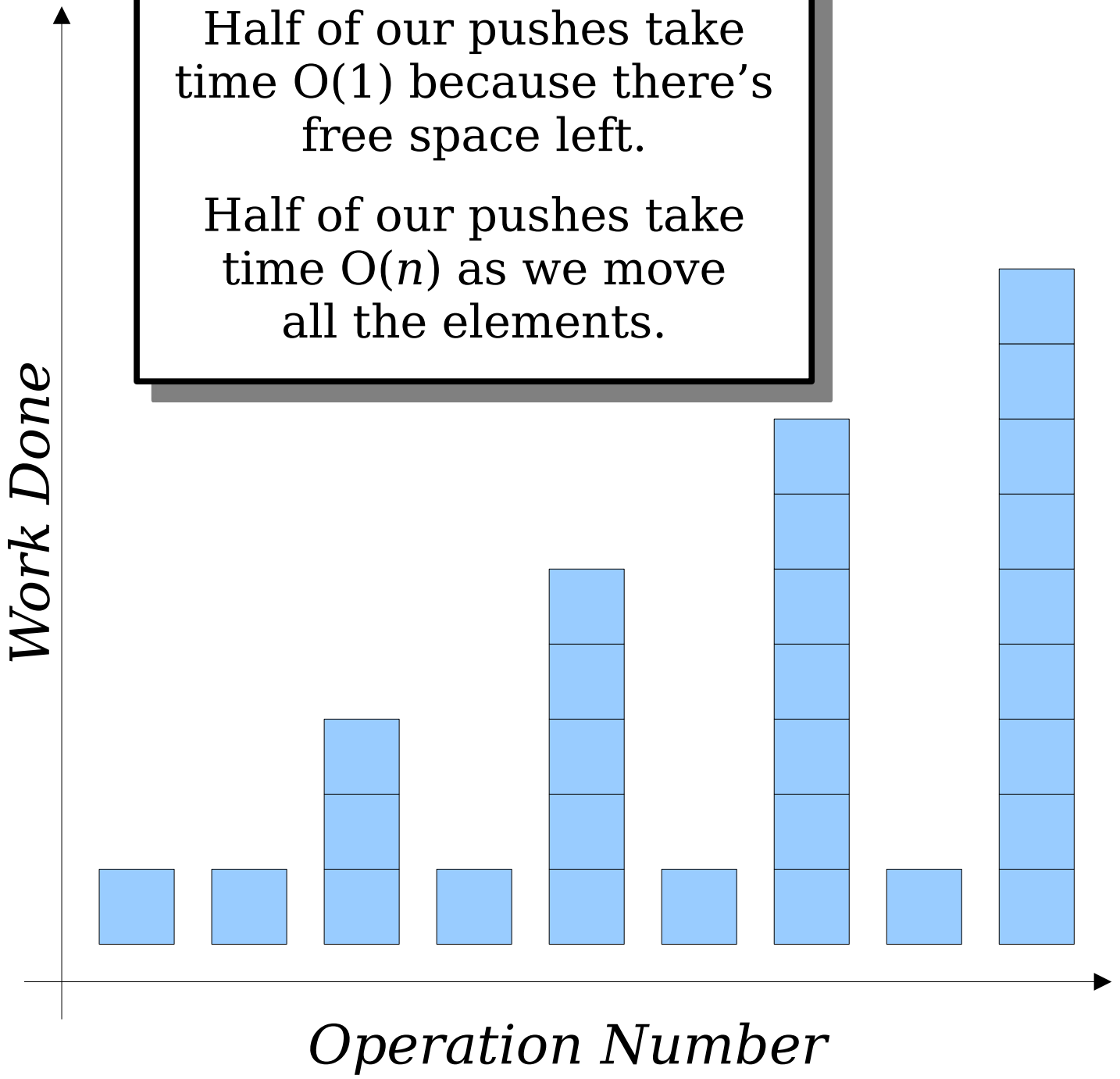
Increase array size by ***adding two***.



Half of our pushes take time O(1) because there's free space left.

Half of our pushes take time O(n) as we move all the elements.

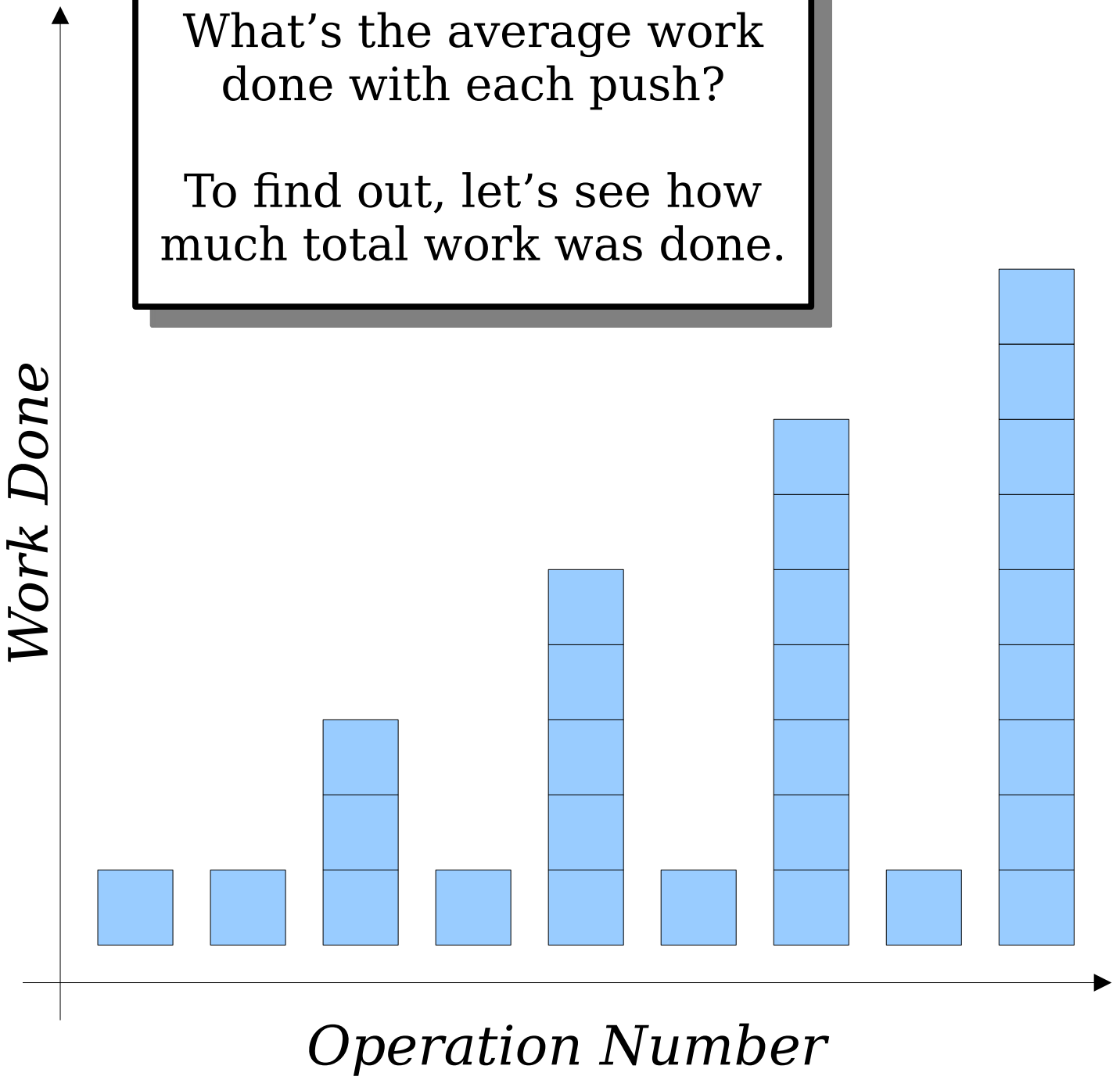
Increase array size by ***adding two***.



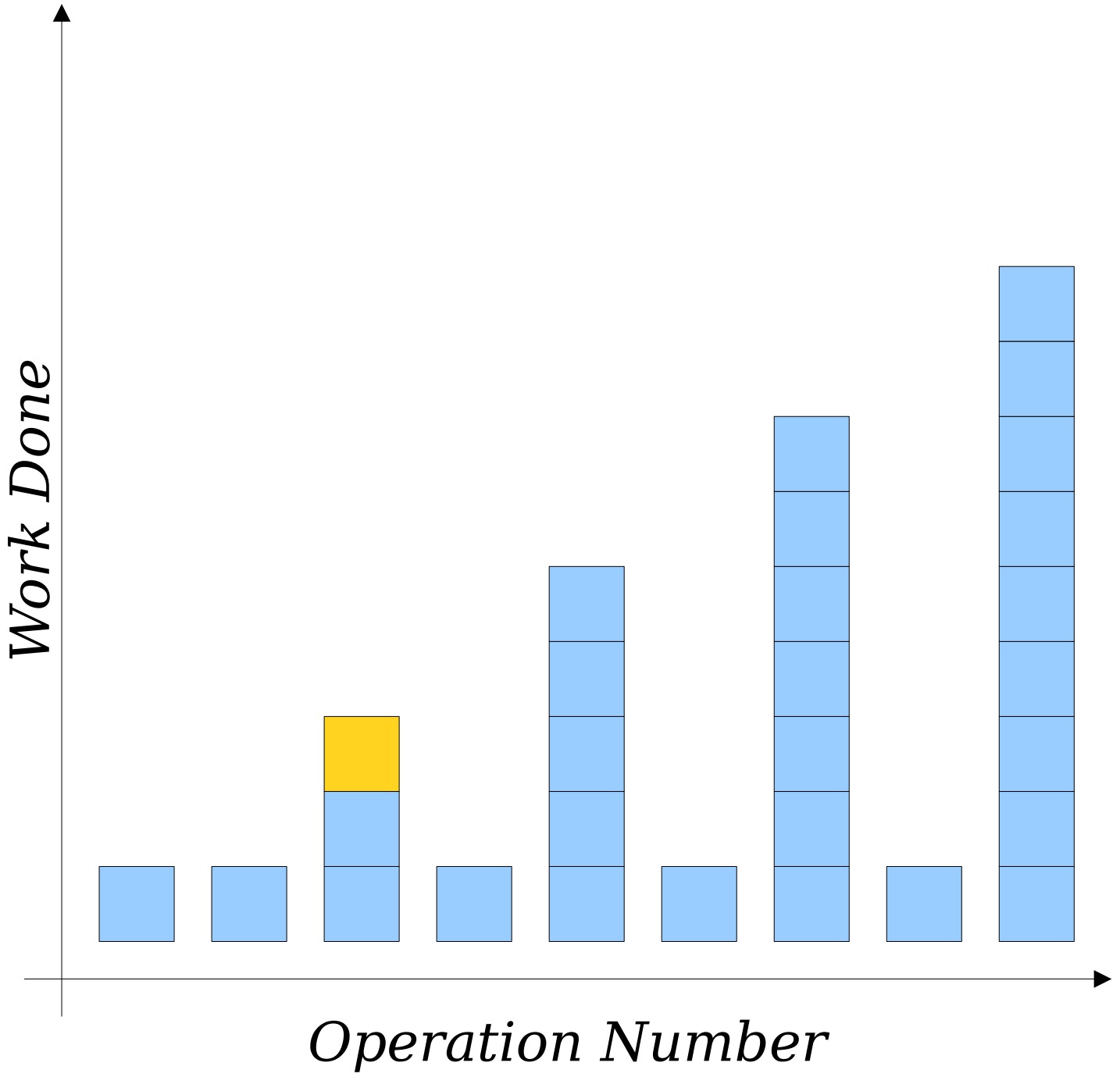
What's the average work done with each push?

To find out, let's see how much total work was done.

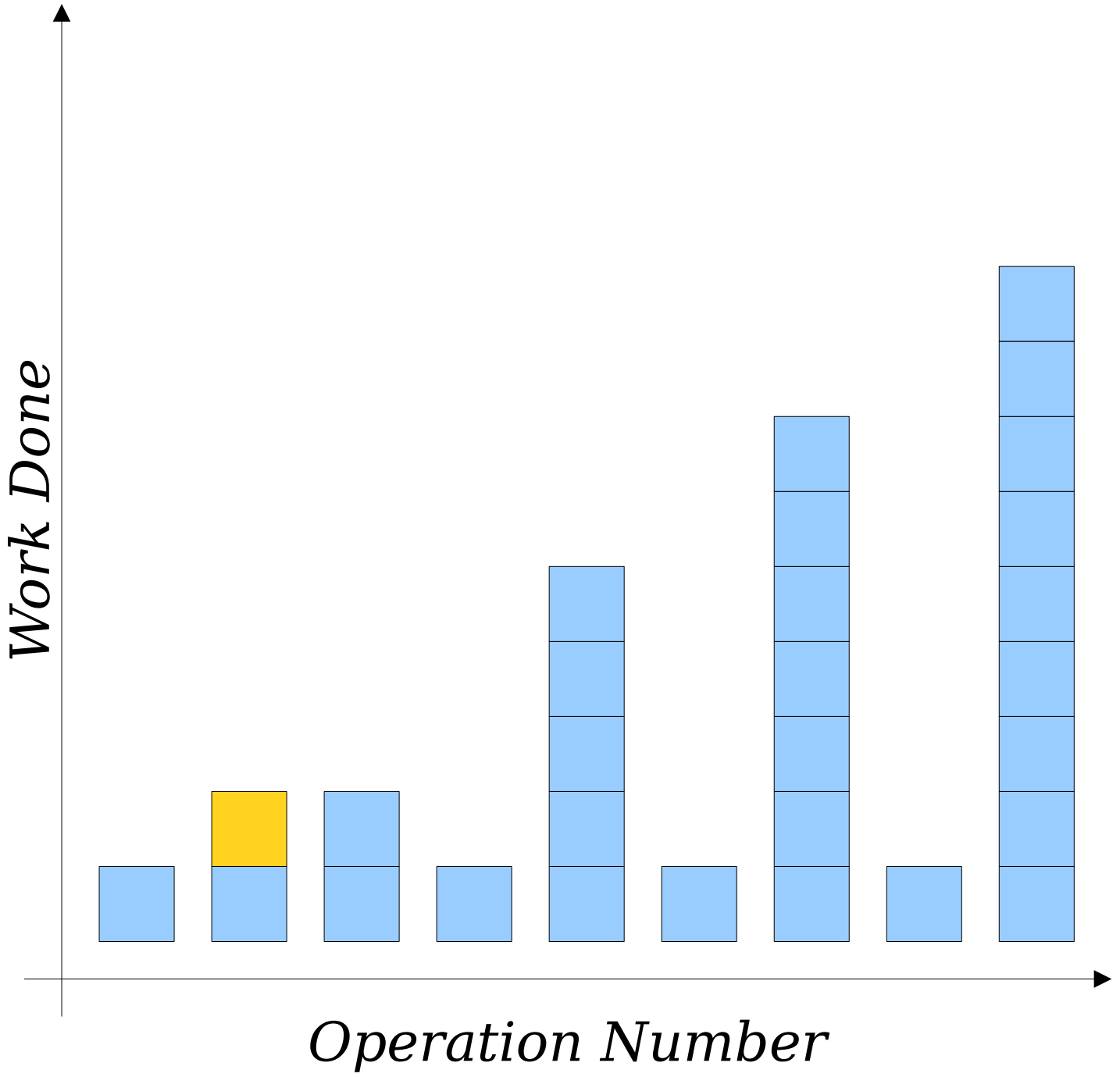
Increase array size by ***adding two***.



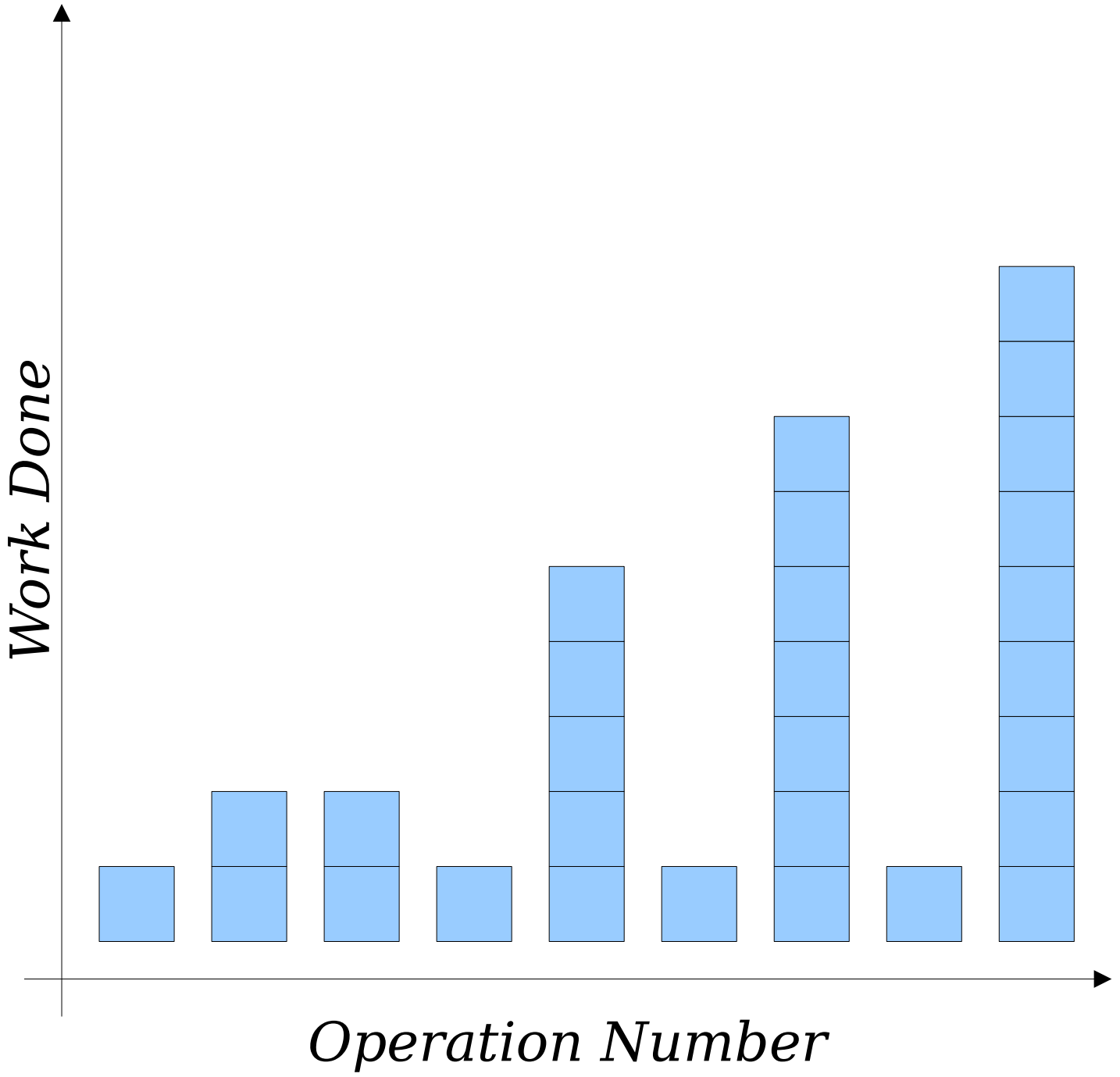
Increase array size by ***adding two***.



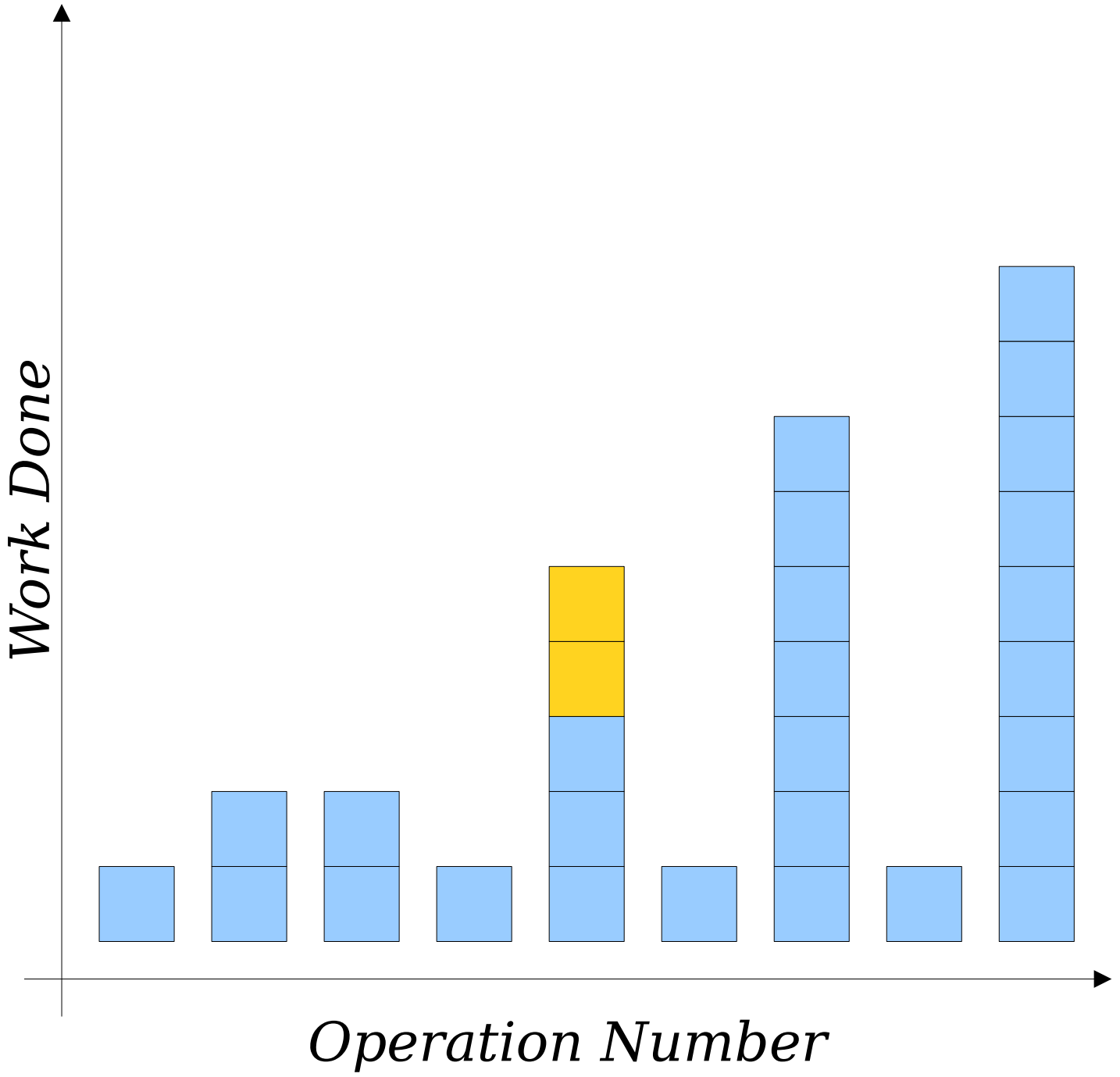
Increase array size by ***adding two***.



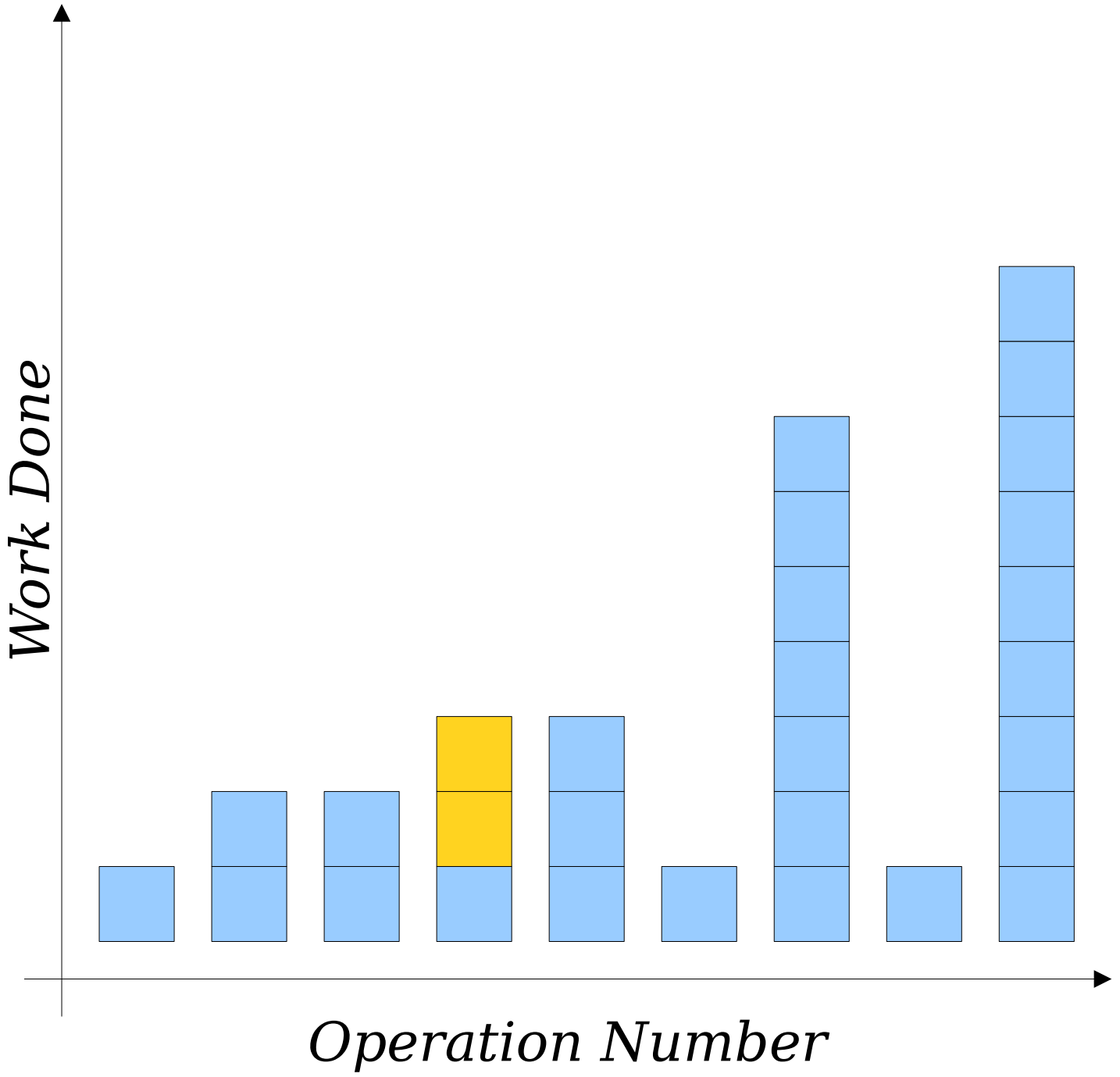
Increase array size by ***adding two***.



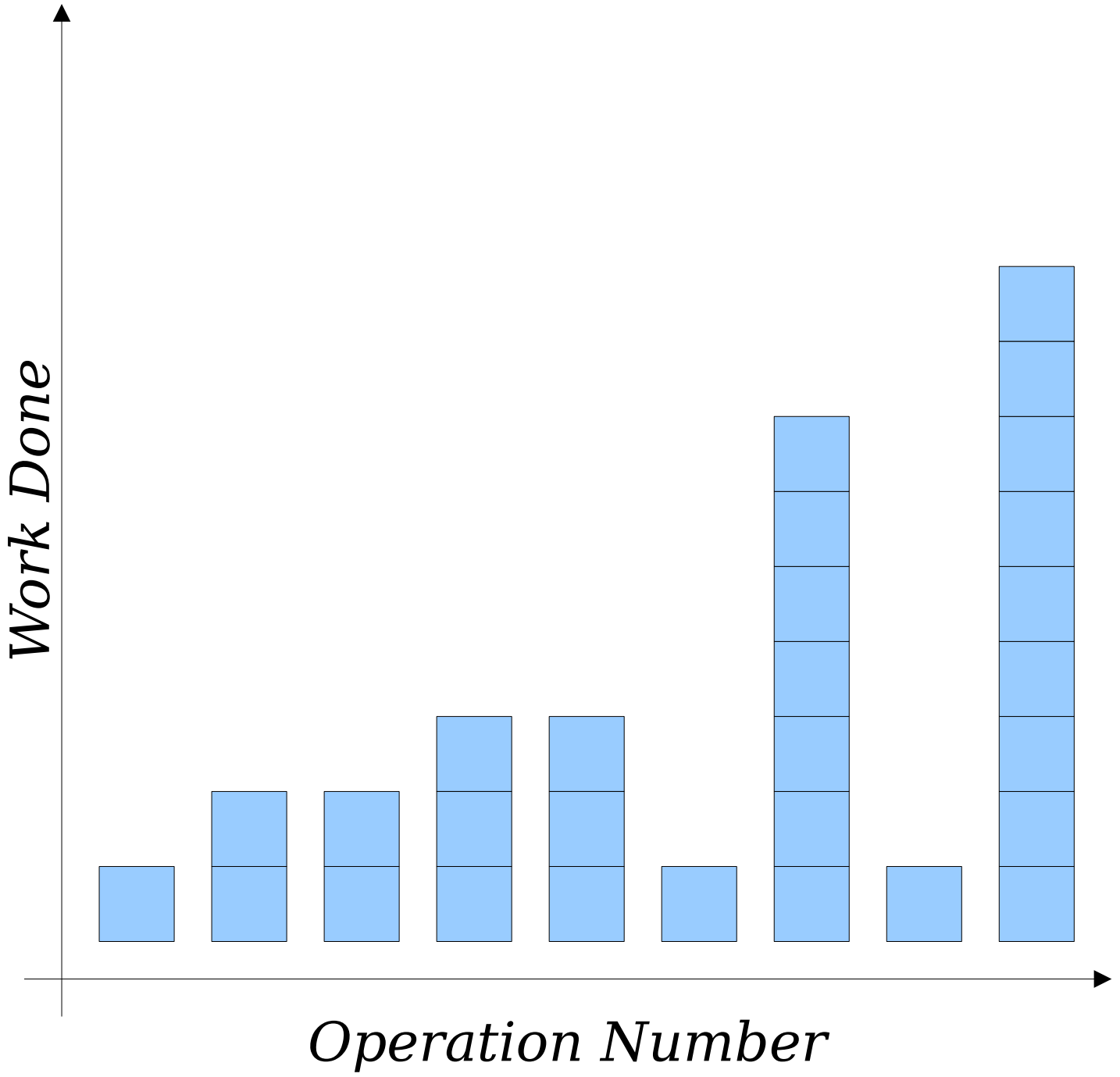
Increase array size by ***adding two***.



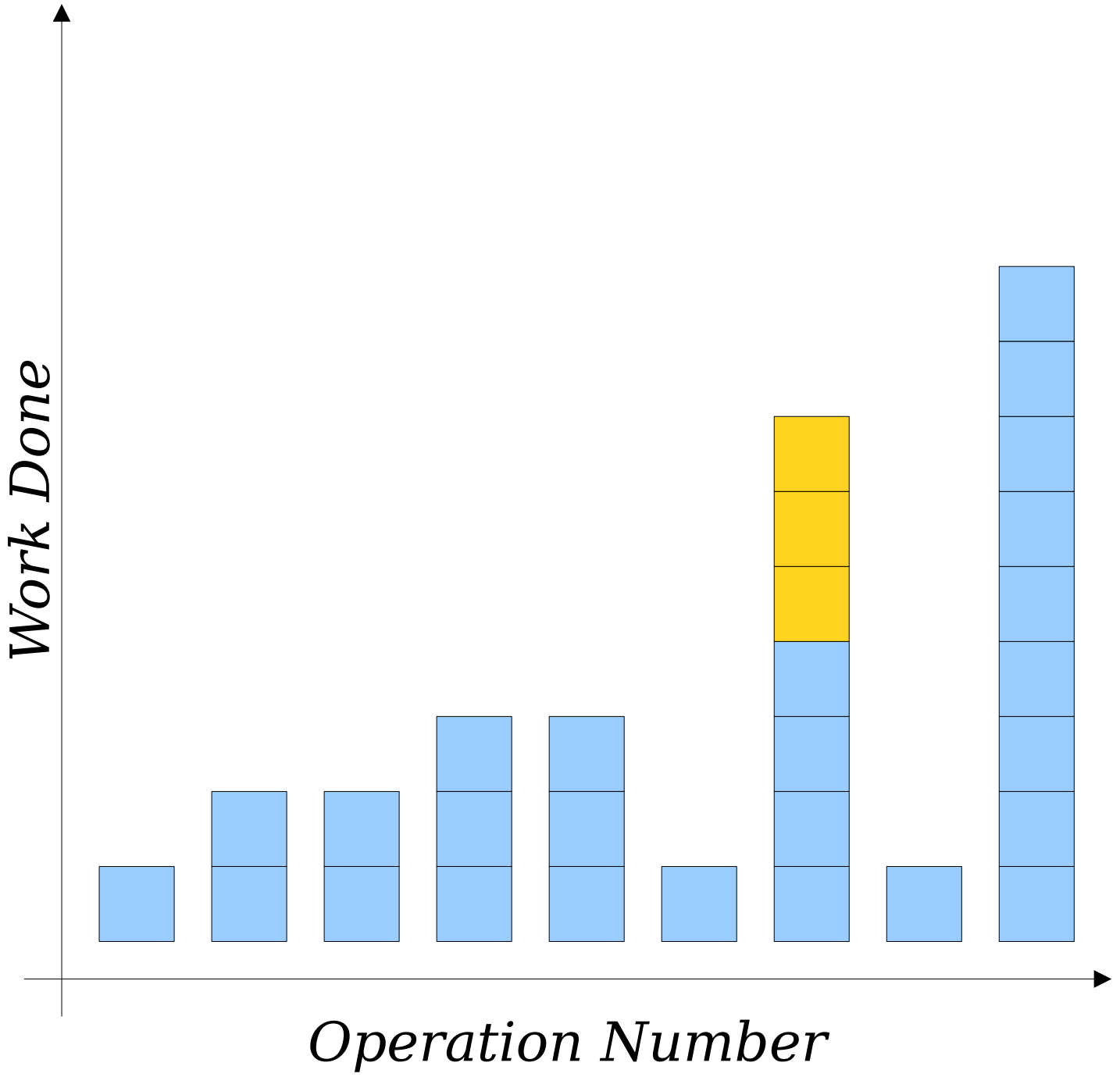
Increase array size by ***adding two***.



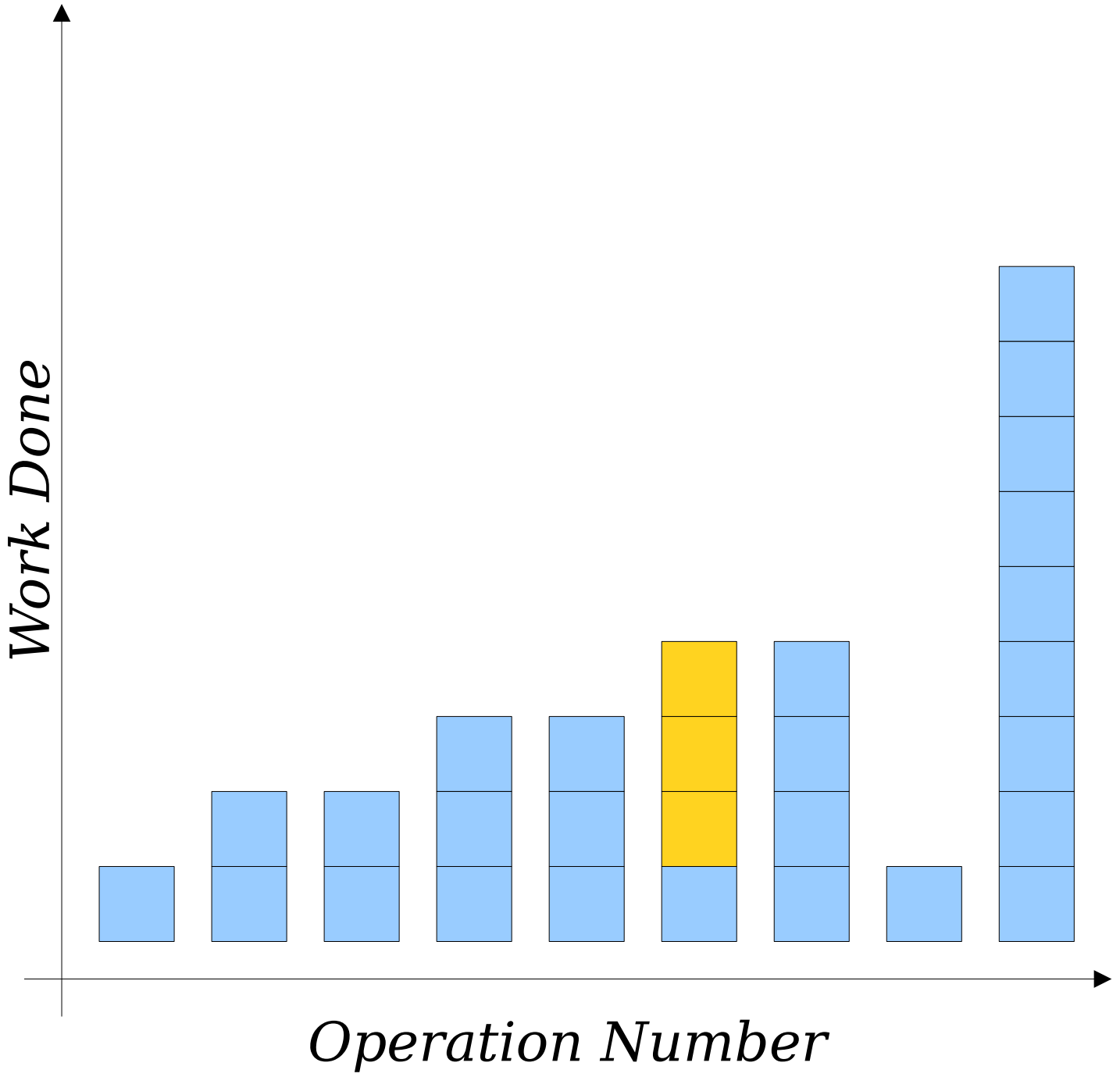
Increase array size by ***adding two***.



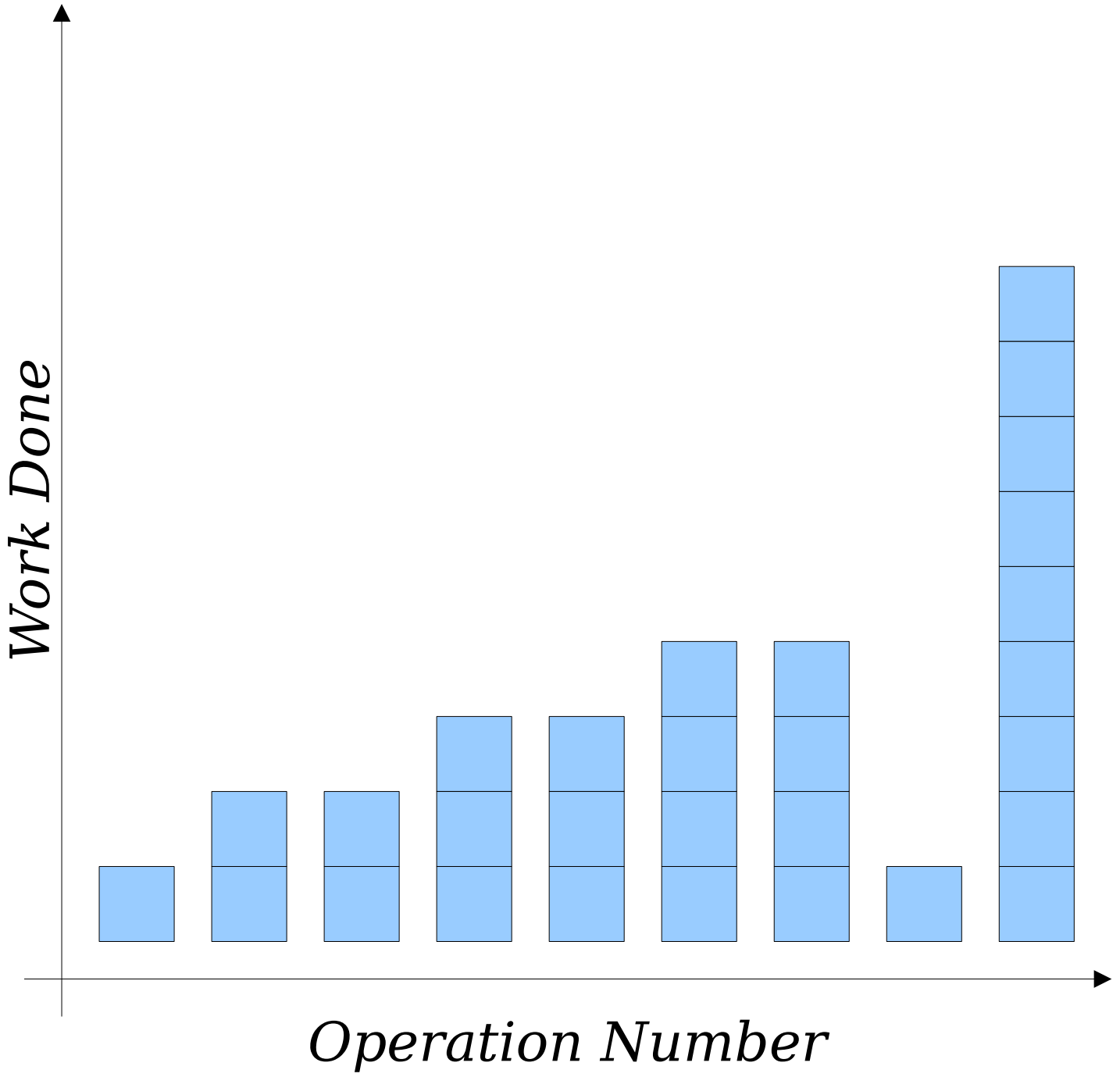
Increase array size by ***adding two***.



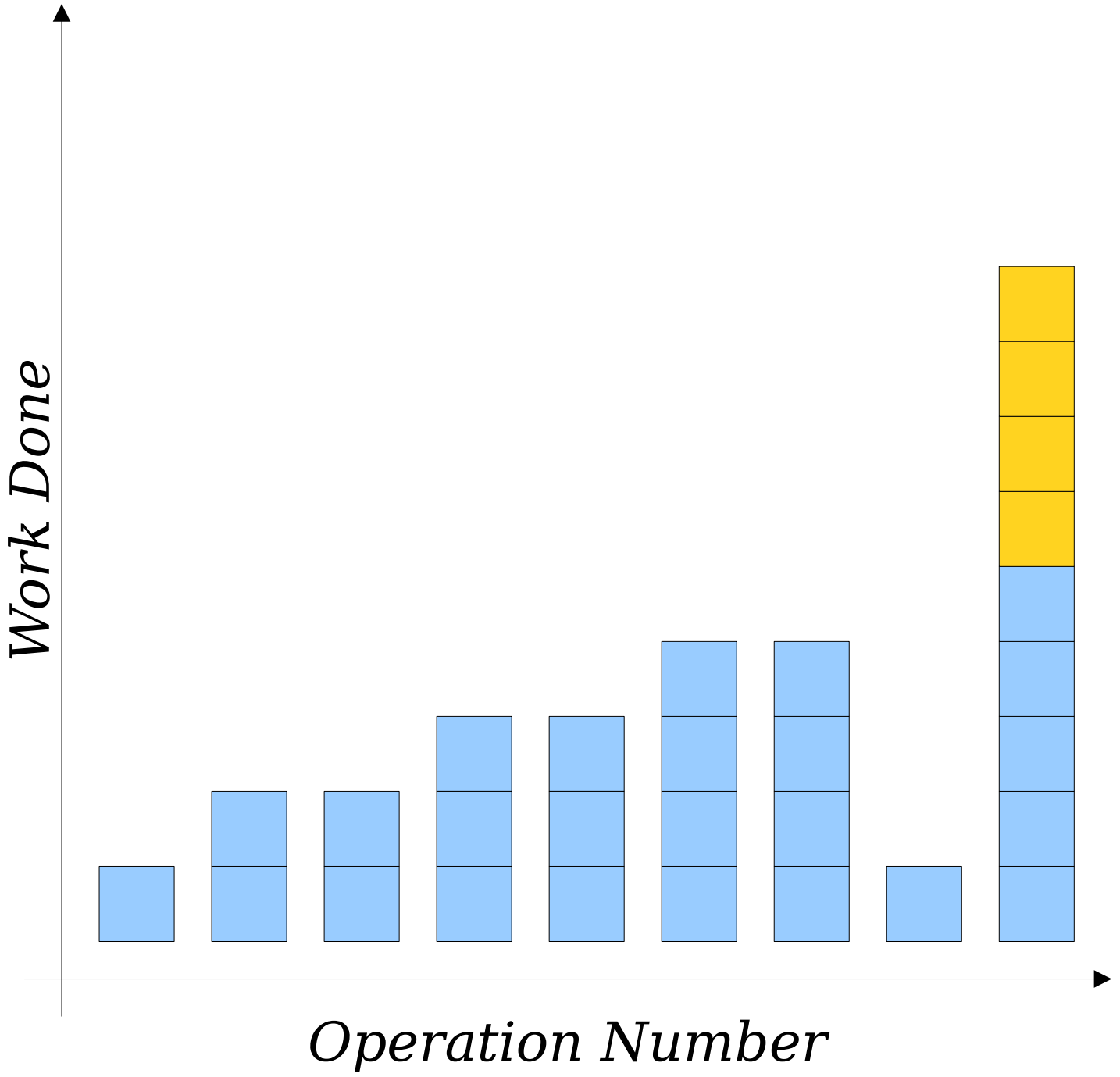
Increase array size by ***adding two***.



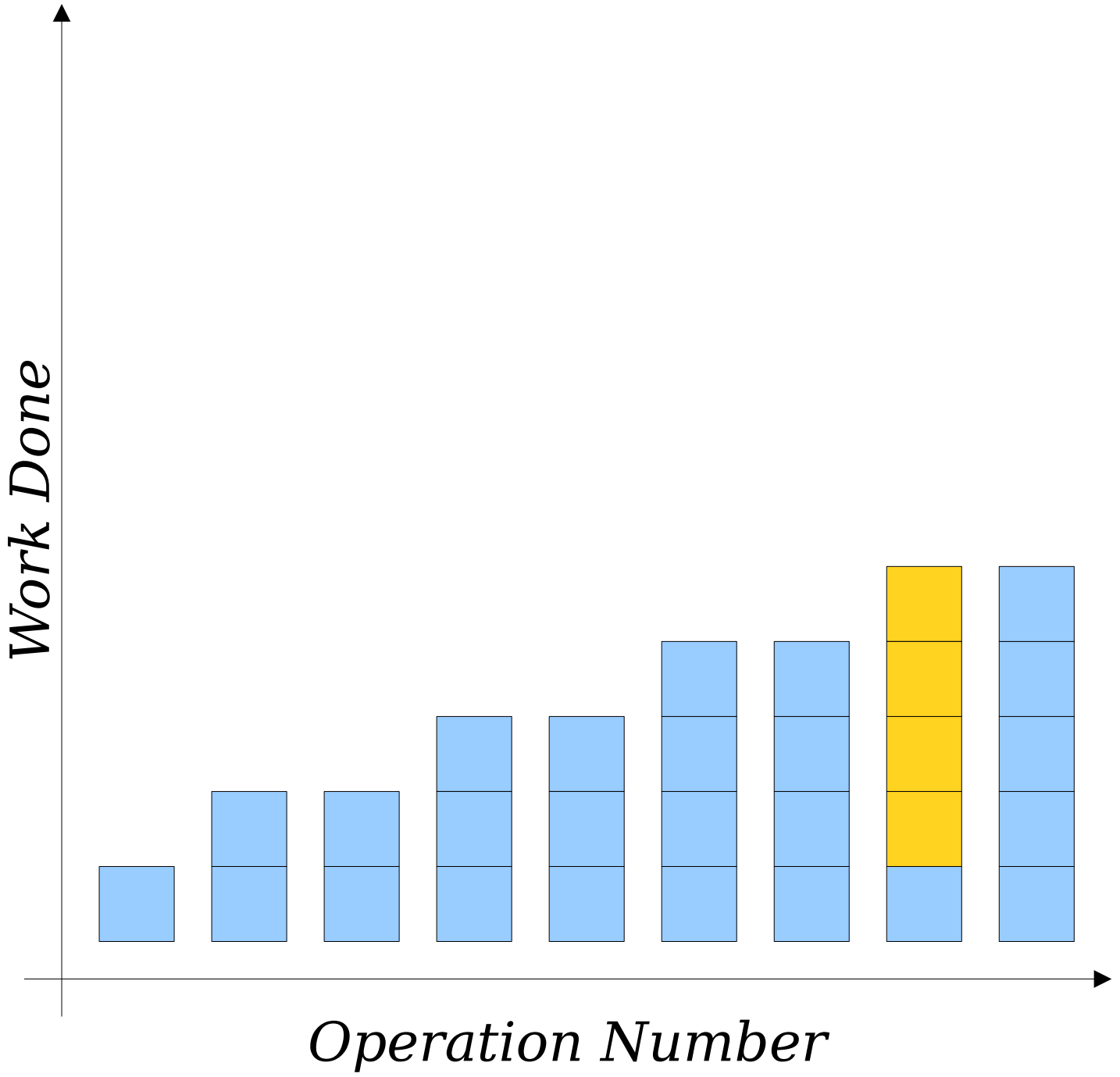
Increase array size by ***adding two***.



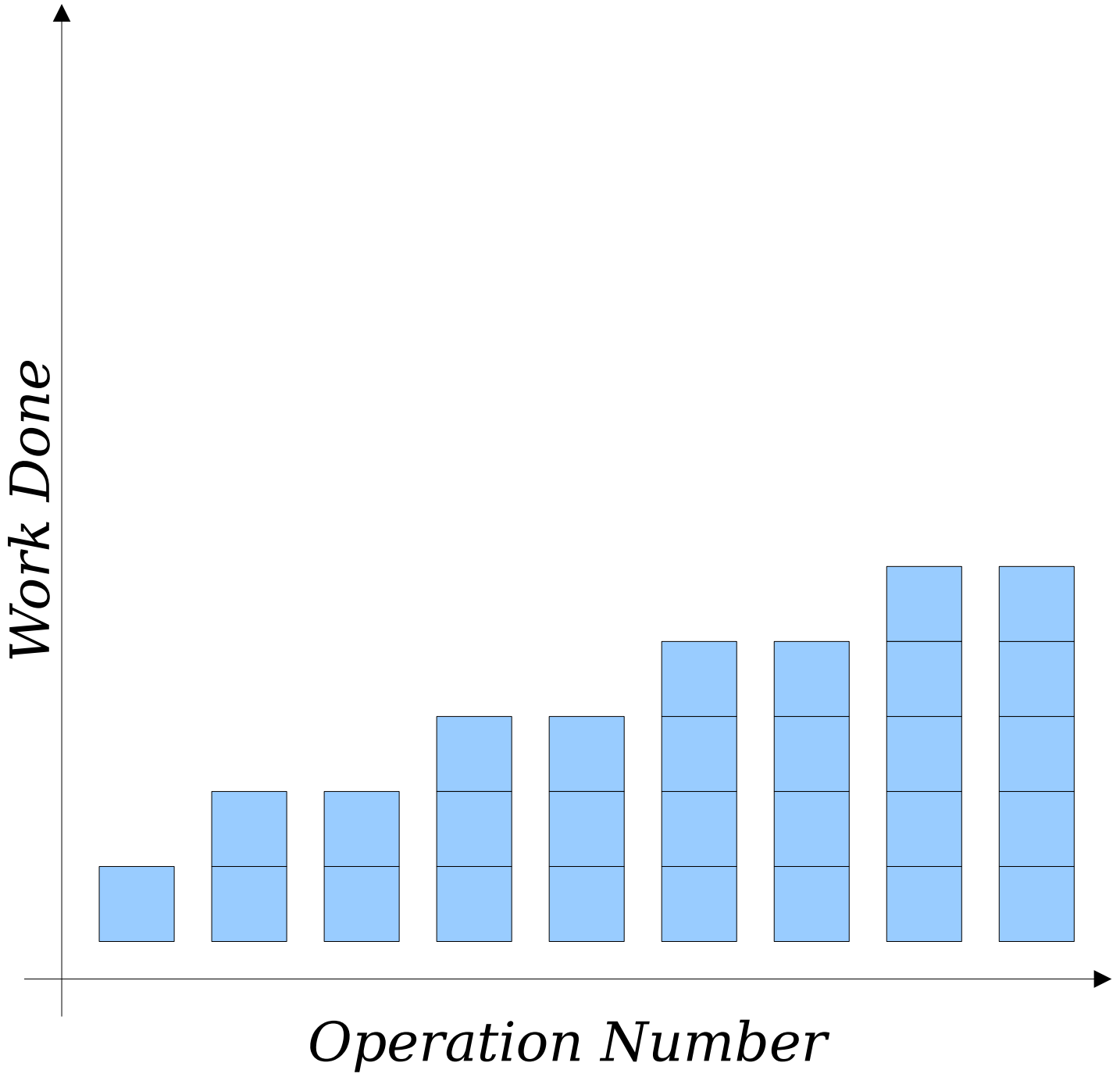
Increase array size by ***adding two***.

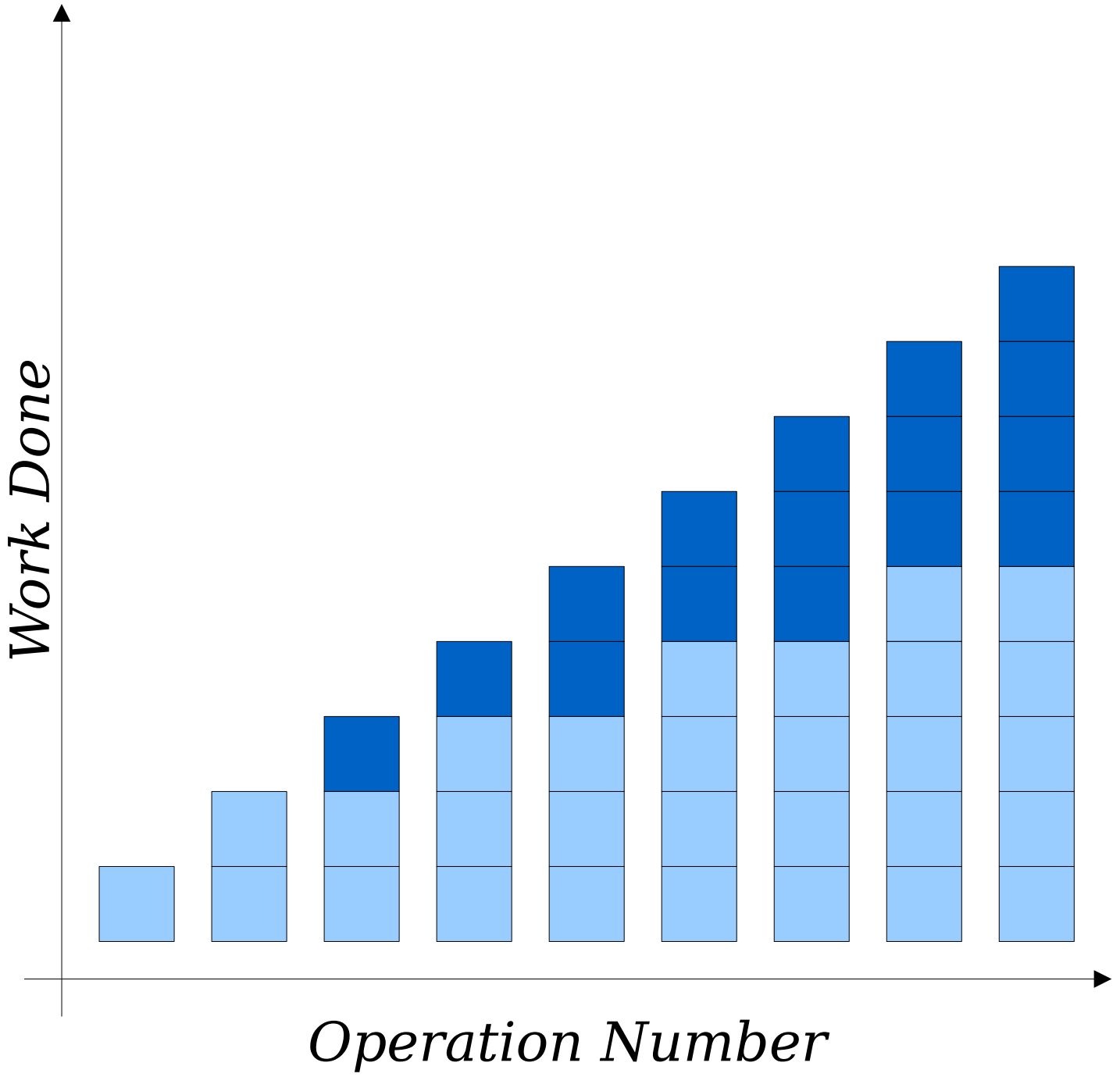


Increase array size by ***adding two***.



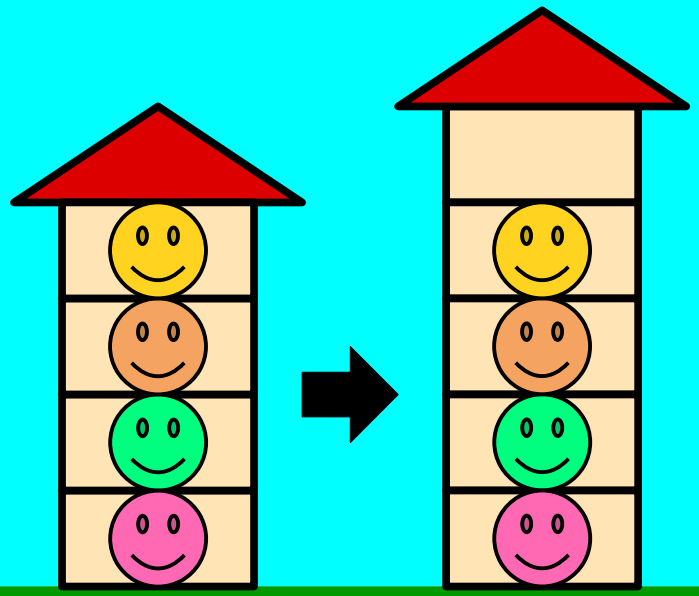
Increase array size by ***adding two***.

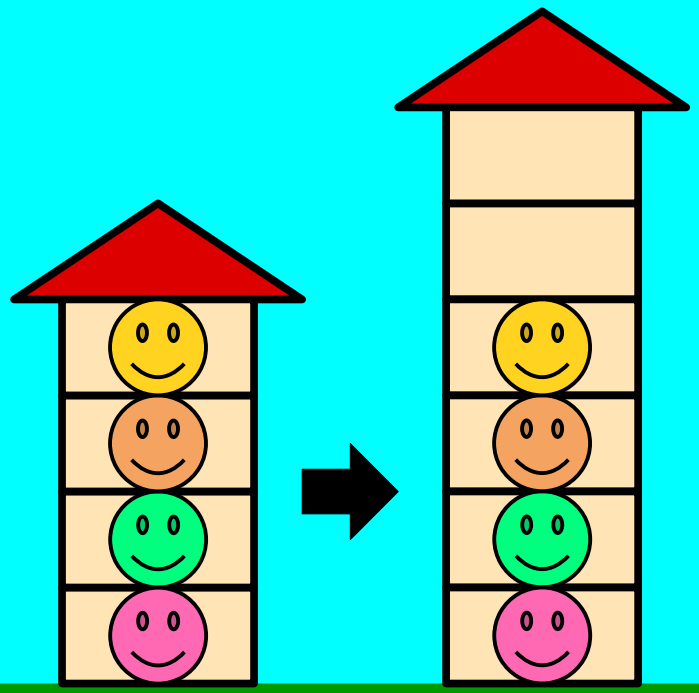


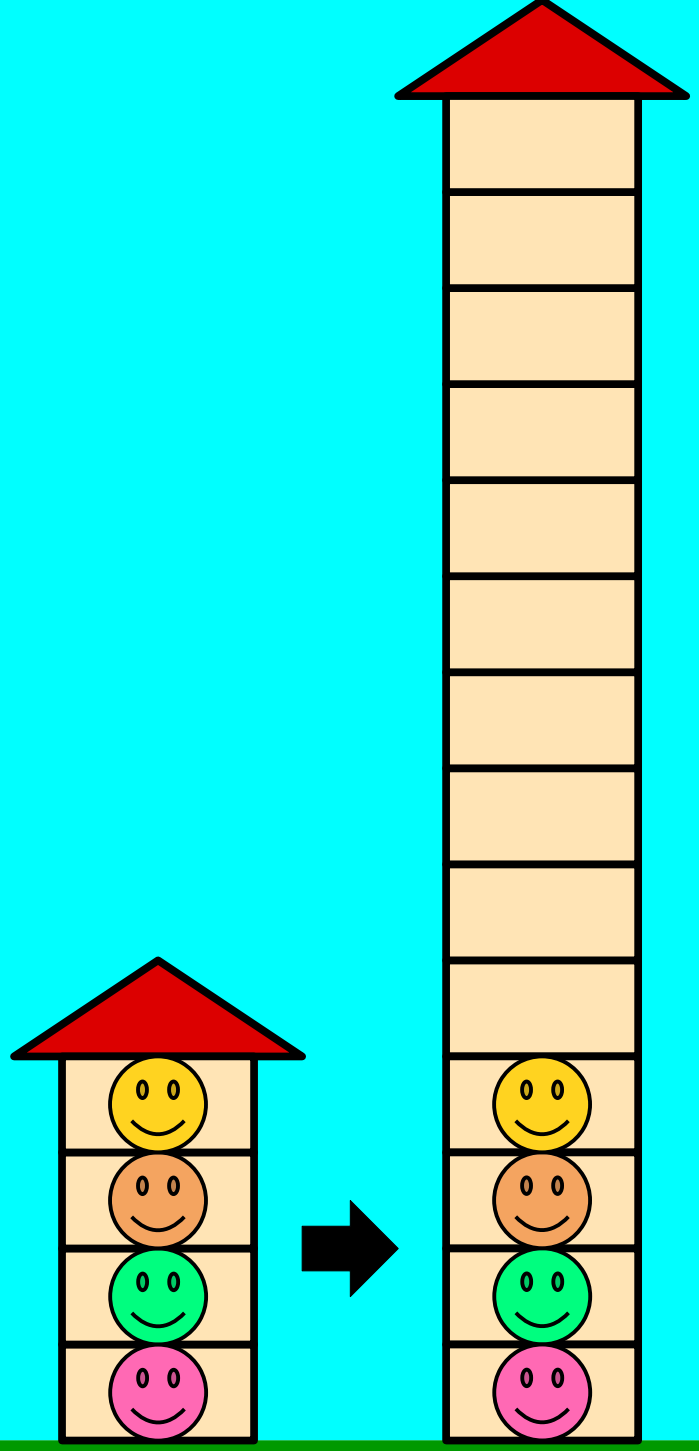


Increase array size by ***adding two***.

This roughly halves the work done.

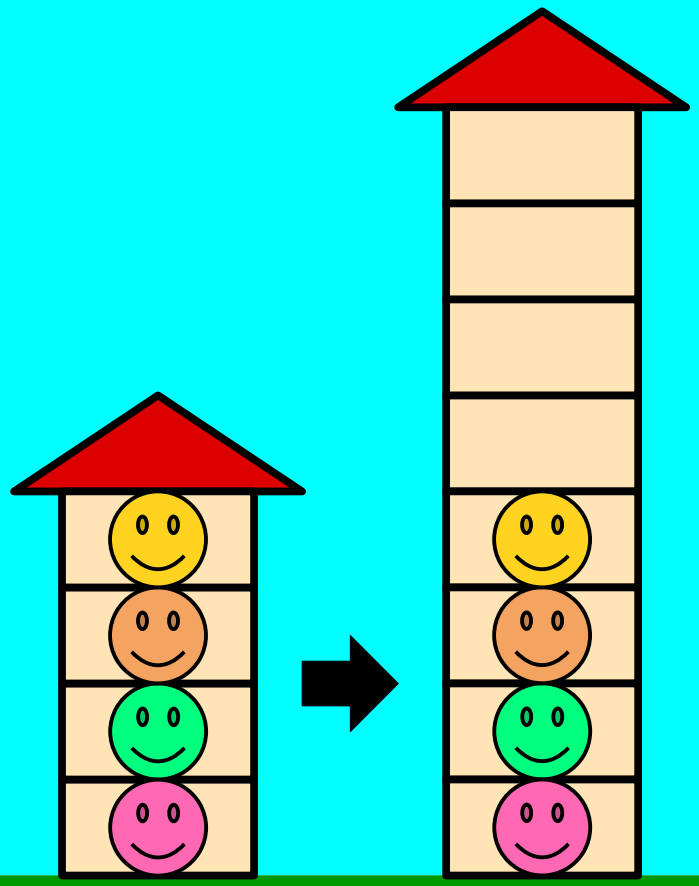






If we make the new array too big, we're might not make use of all the new space.

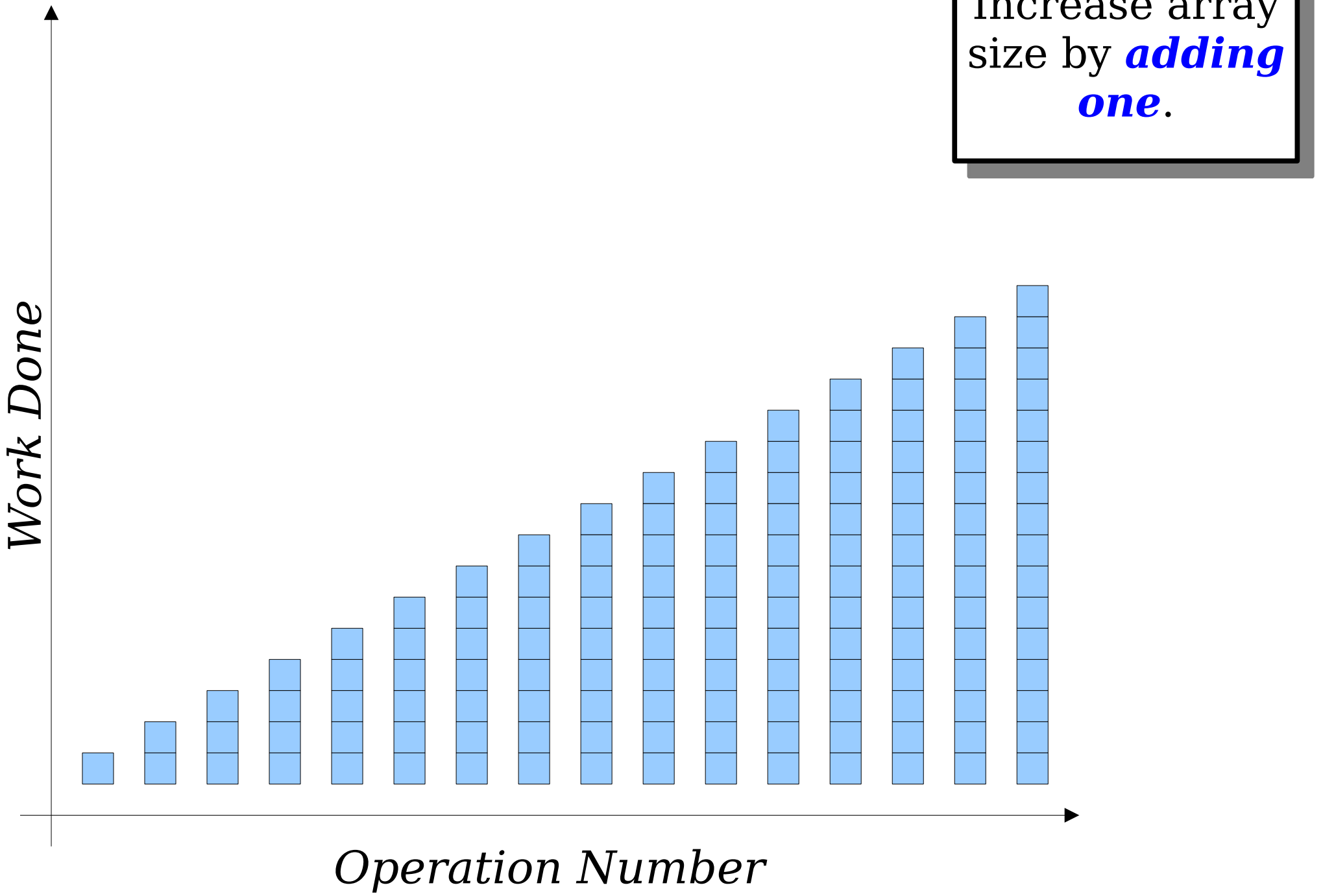
What's a good compromise?



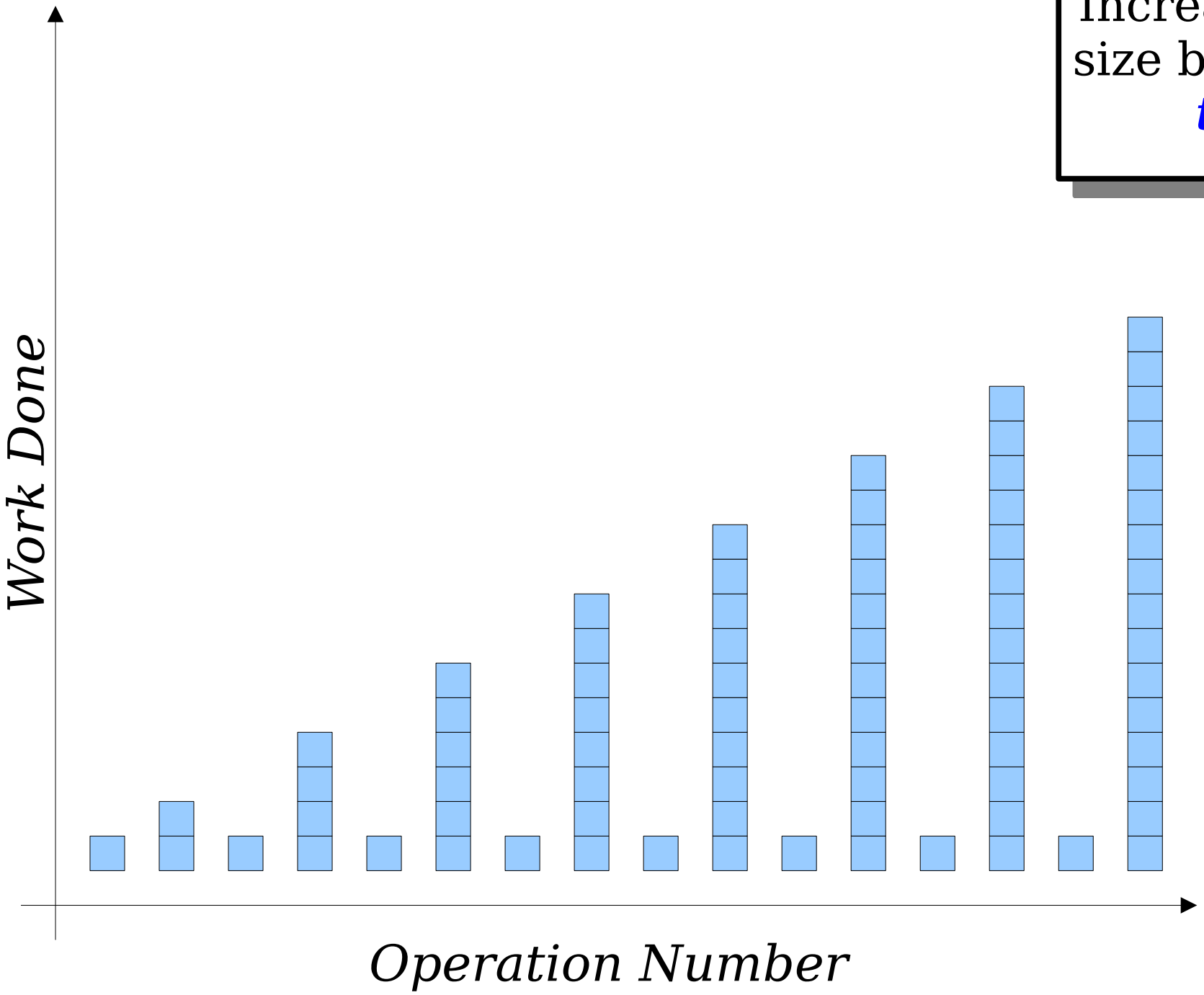
Idea: Make the new array twice as big as the old one.

This gives us a lot of free space, and we never use more than twice the space we need.

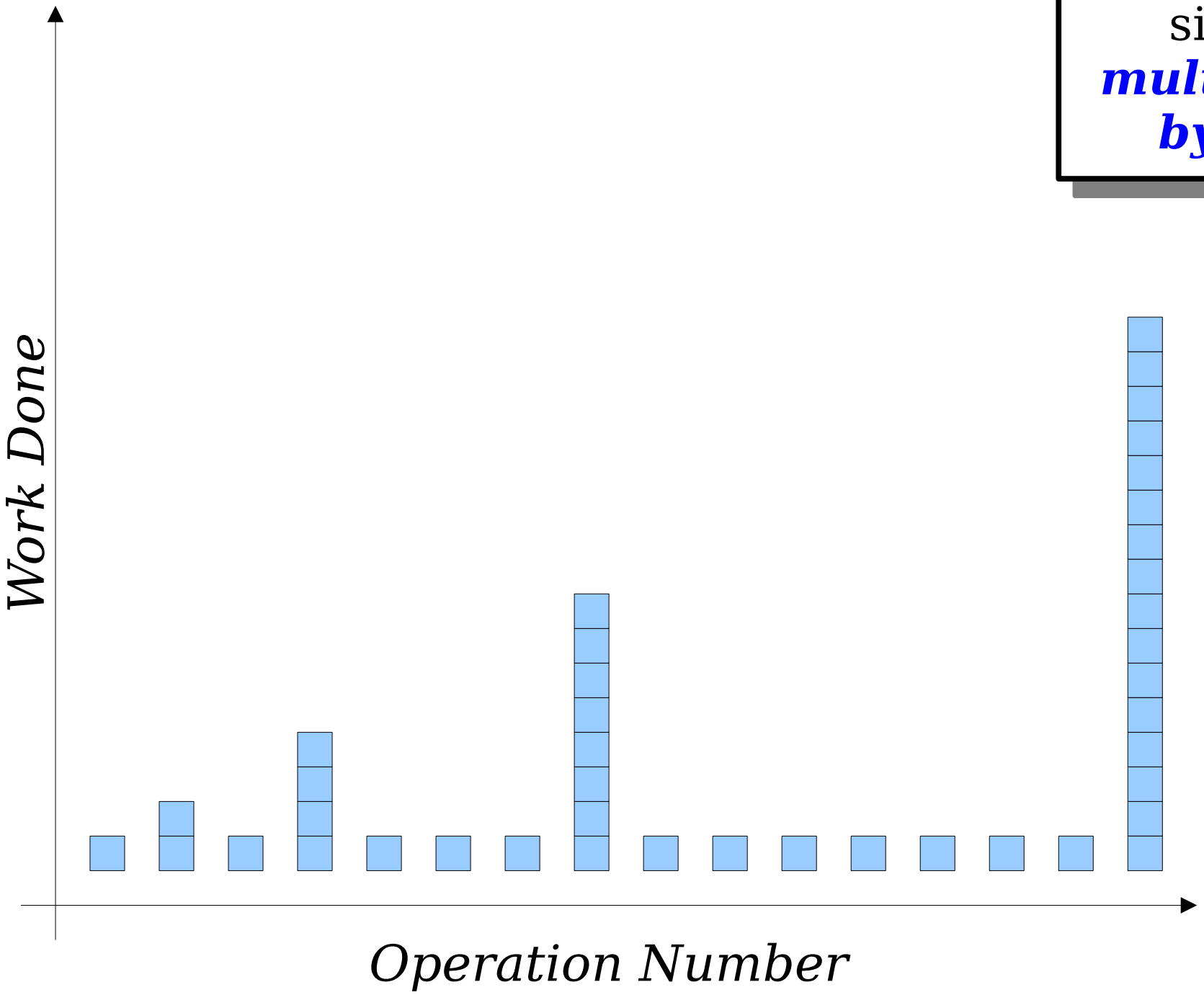
How do we analyze this?


Increase array size by adding one.
Work Done
Operation Number

Increase array size by ***adding two***.



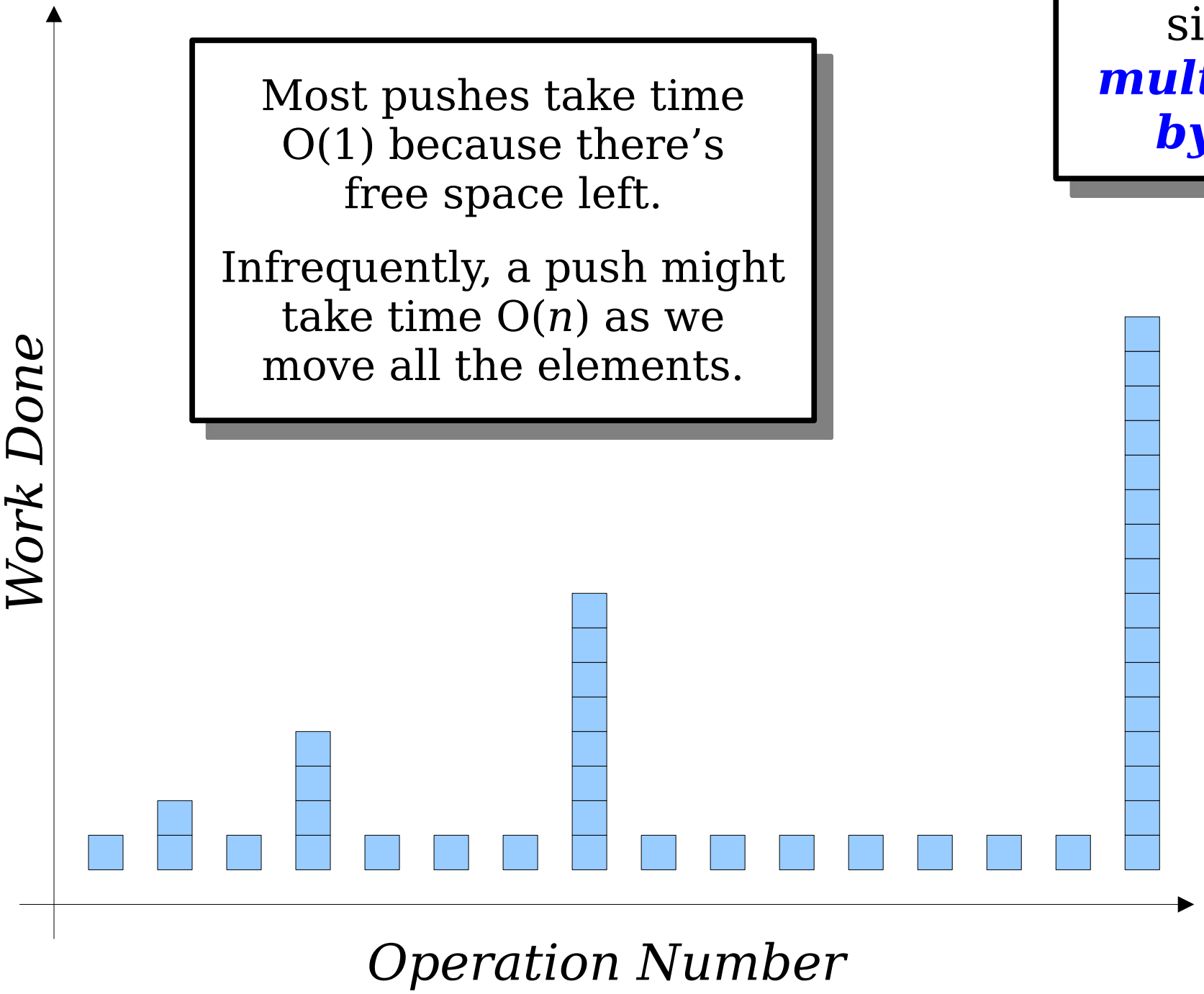
Increase array size by ***multiplying by two***.



Increase array size by ***multiplying by two***.

Most pushes take time O(1) because there's free space left.

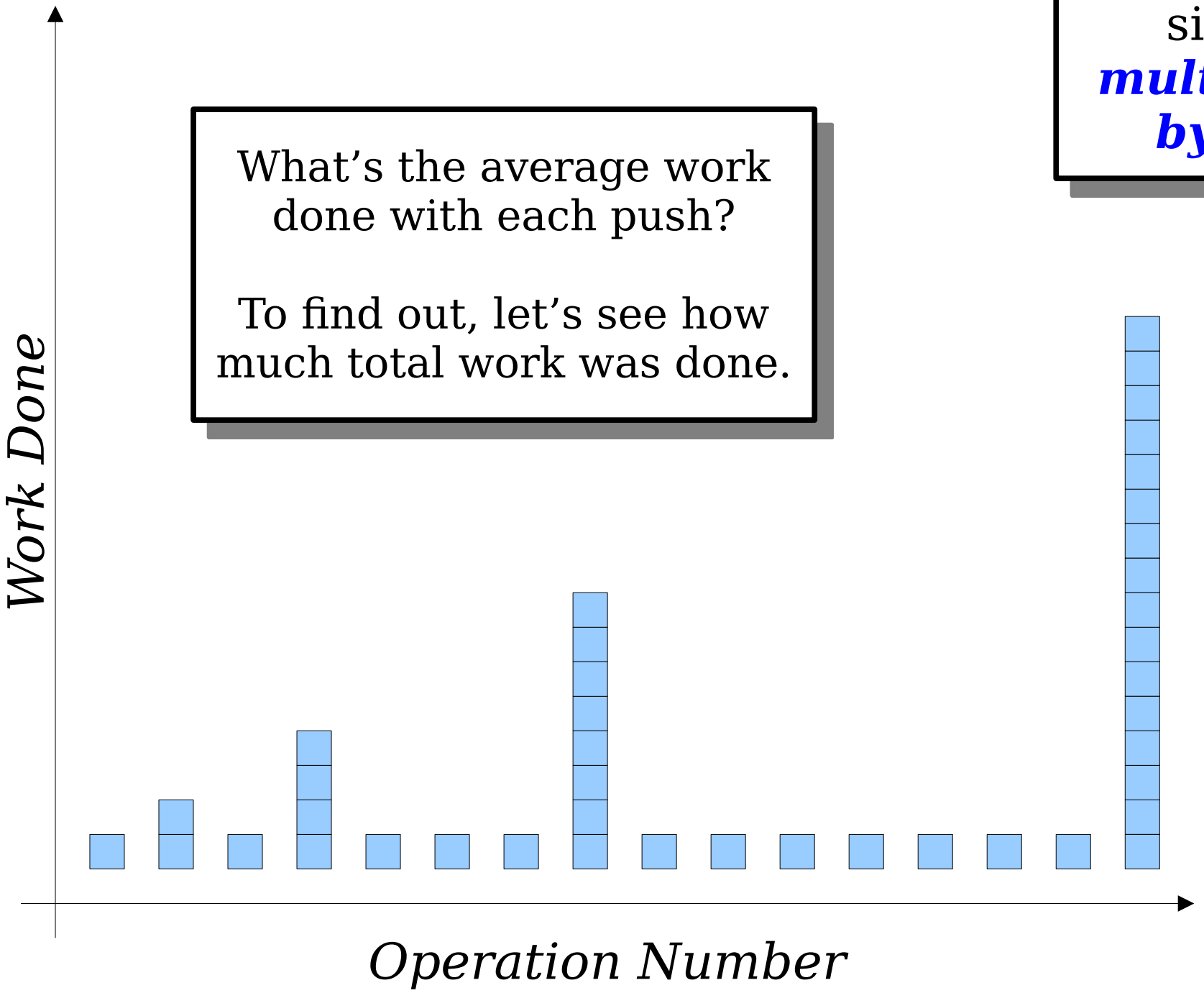
Infrequently, a push might take time O(n) as we move all the elements.



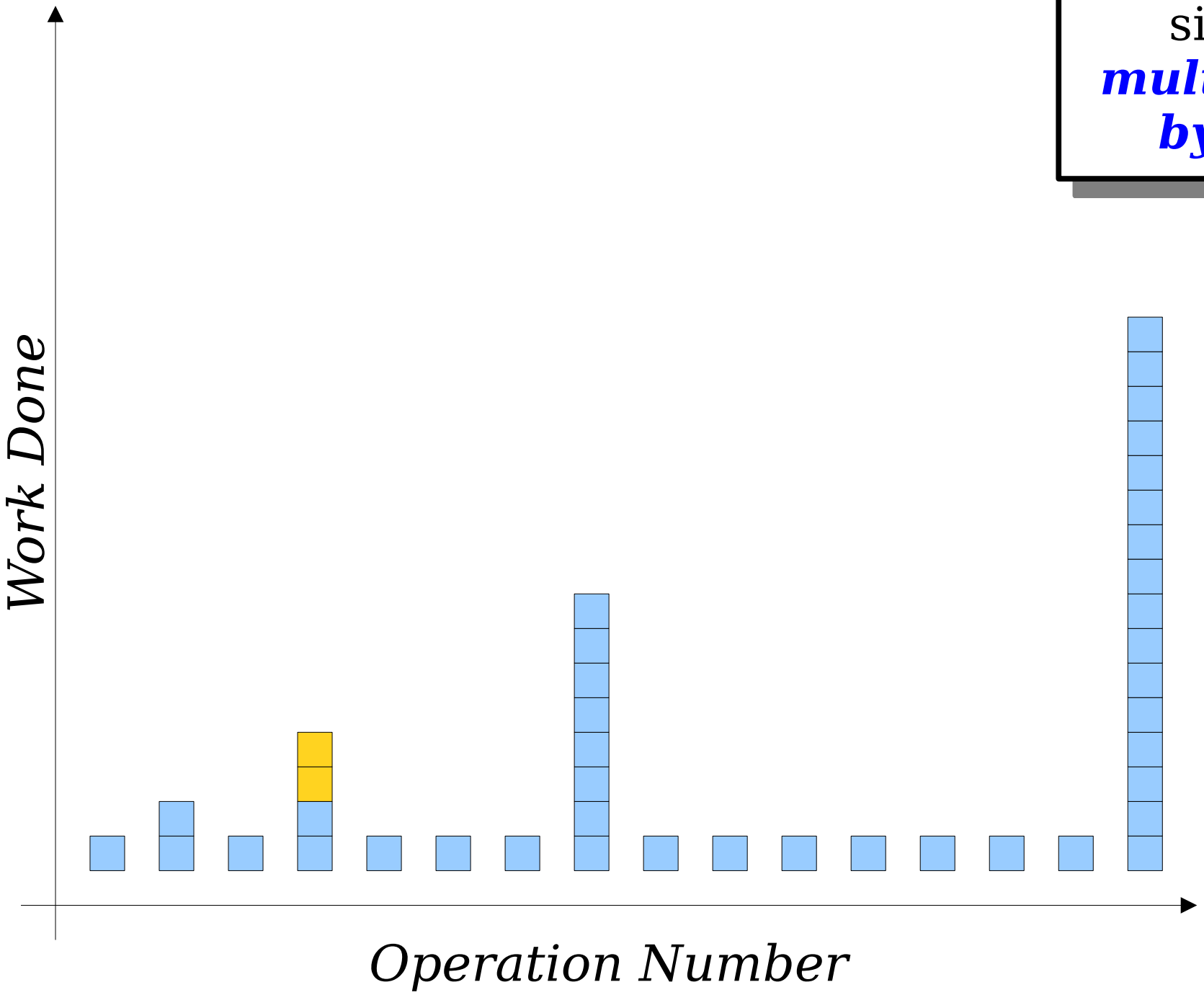
Increase array size by ***multiplying by two***.

What's the average work done with each push?

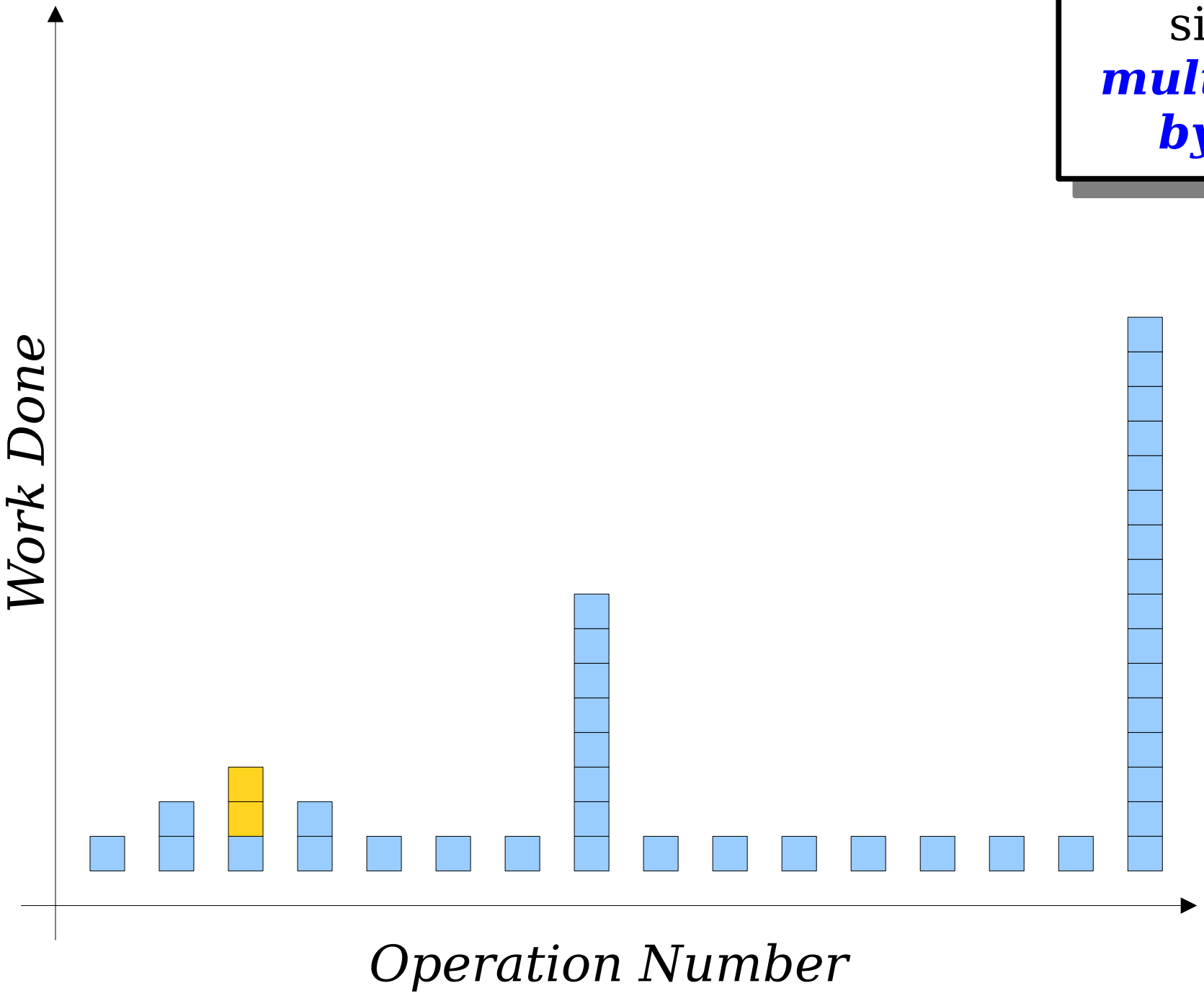
To find out, let's see how much total work was done.



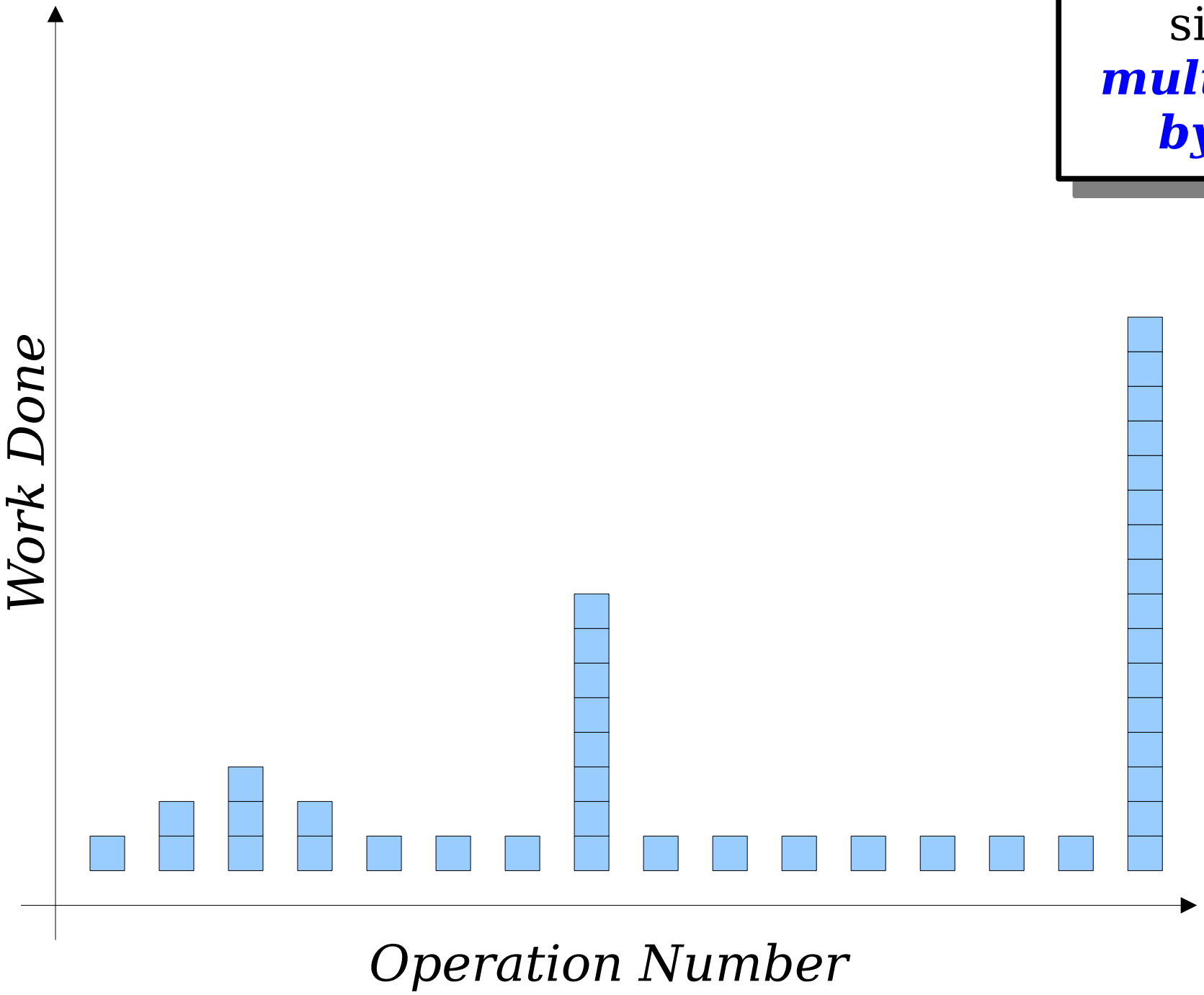
Increase array size by ***multiplying by two***.



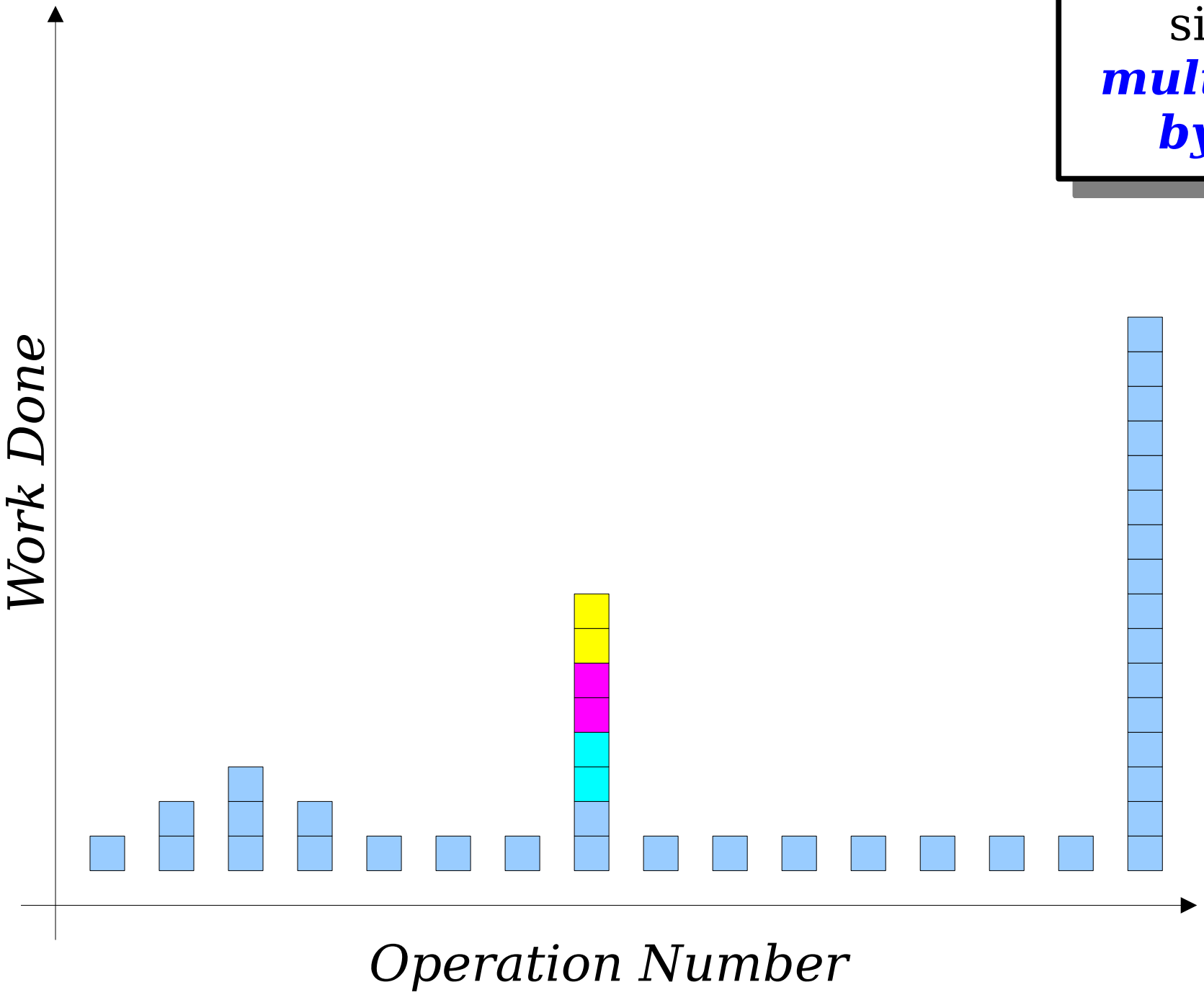
Increase array size by ***multiplying by two***.



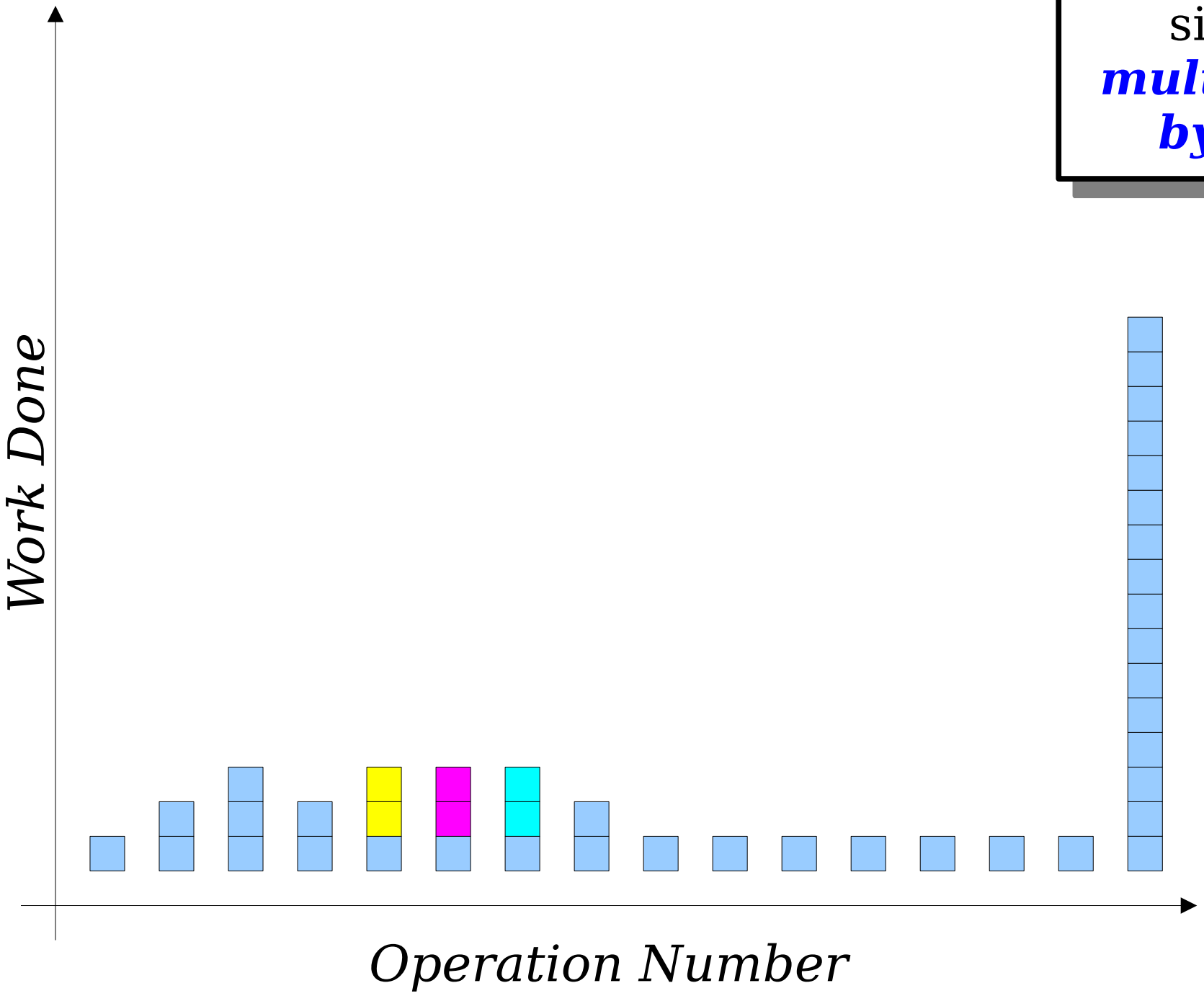
Increase array size by ***multiplying by two***.



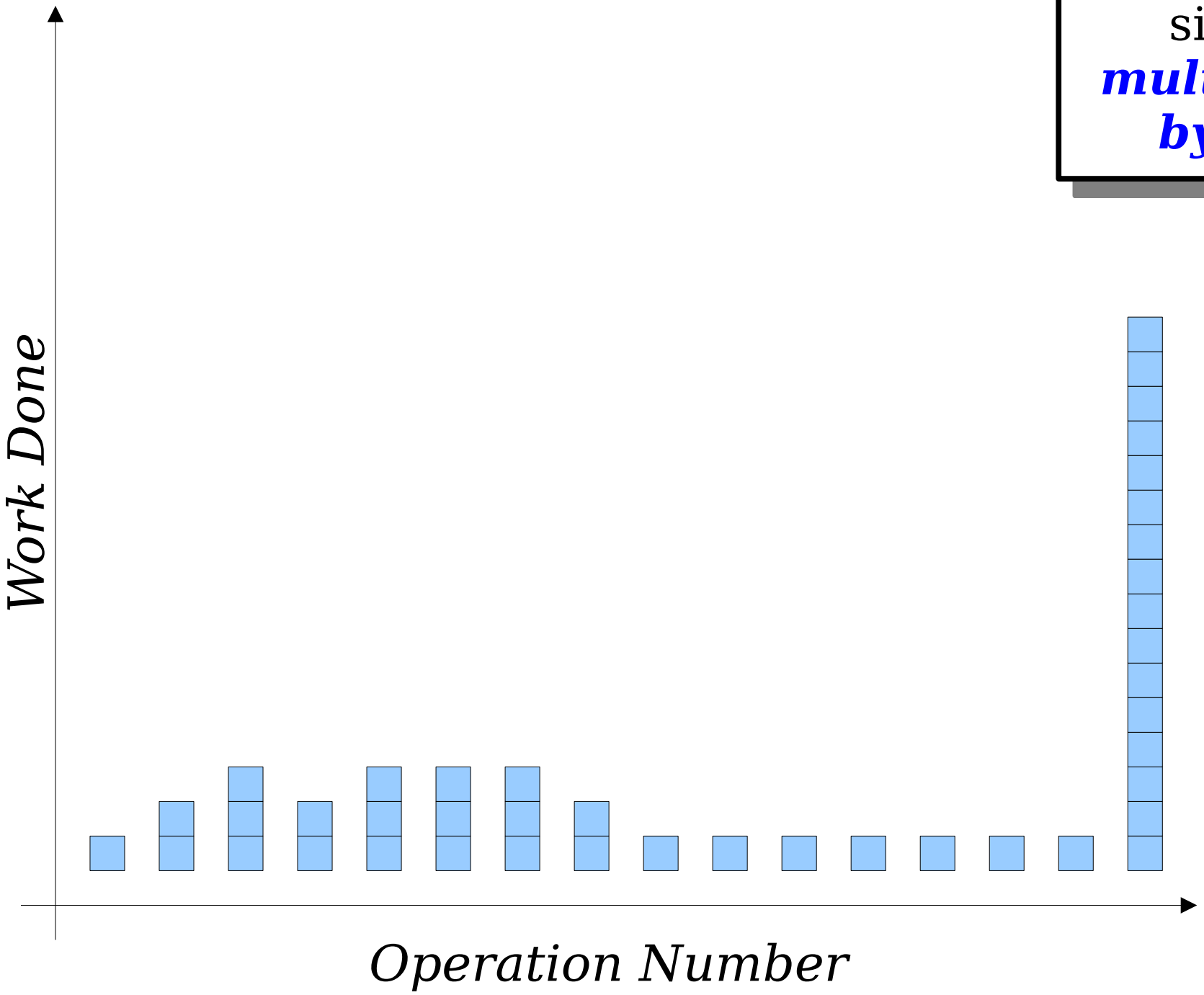
Increase array size by ***multiplying by two***.



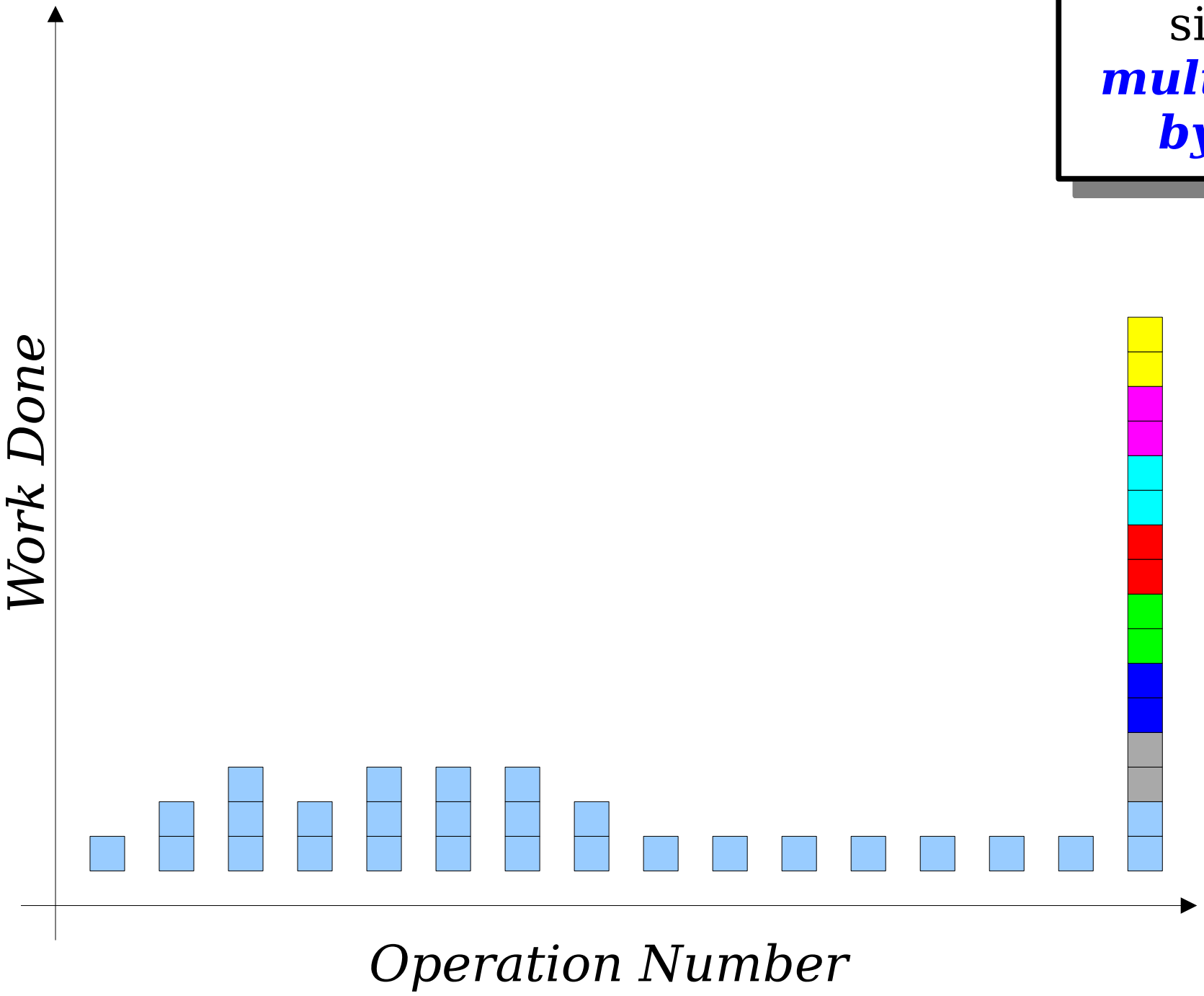
Increase array size by ***multiplying by two***.



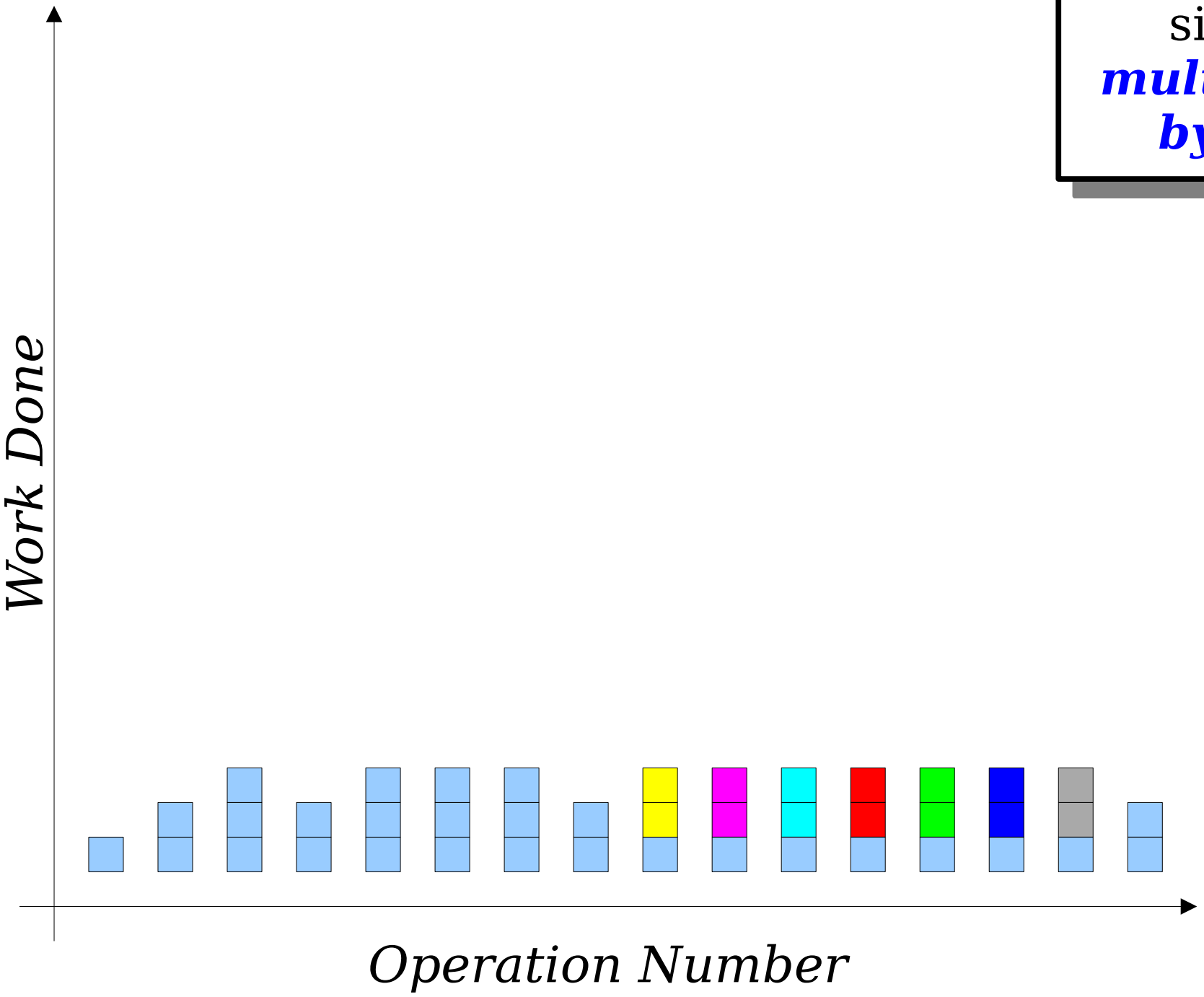
Increase array size by ***multiplying by two***.



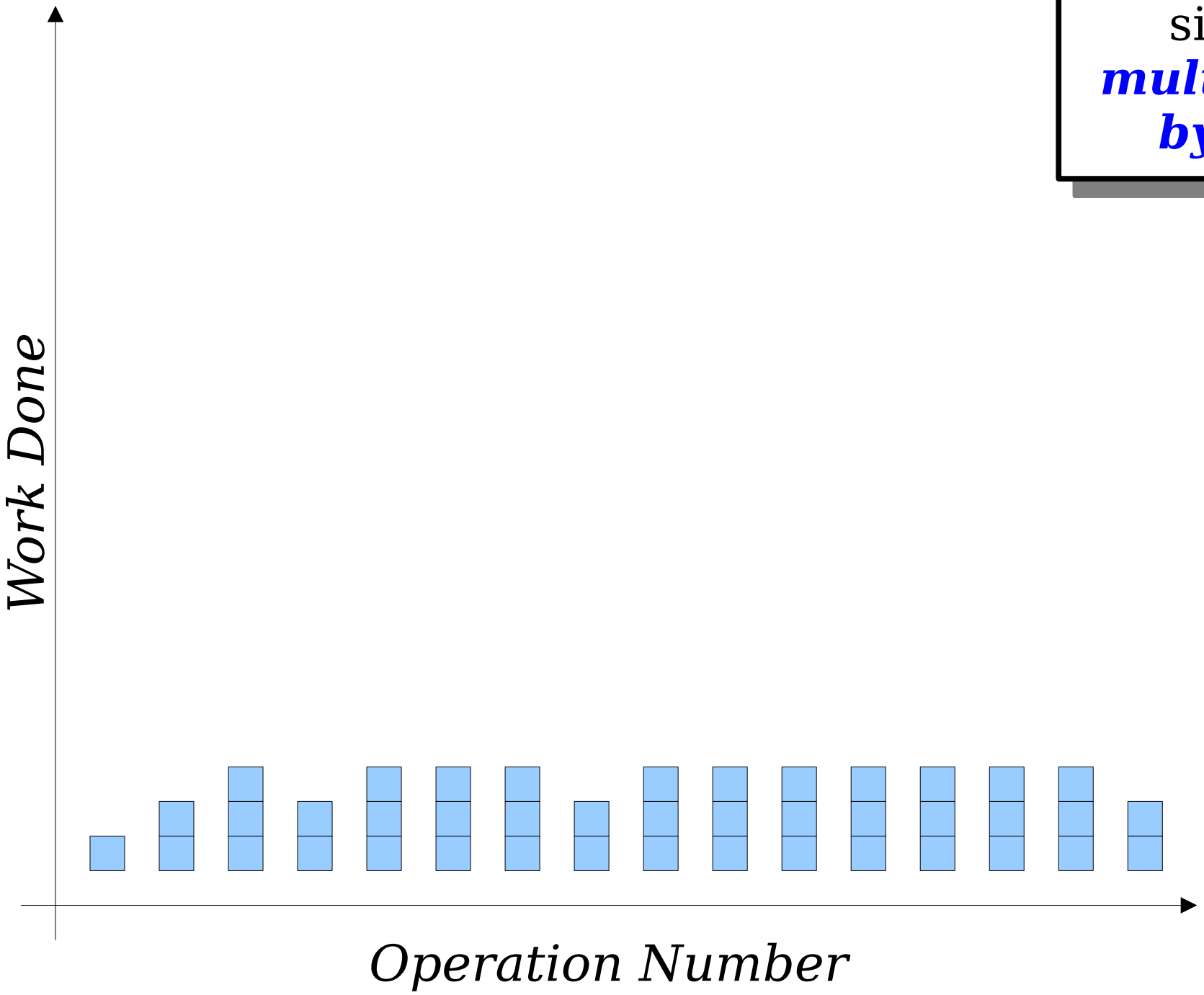
Increase array size by ***multiplying by two***.



Increase array size by ***multiplying by two***.



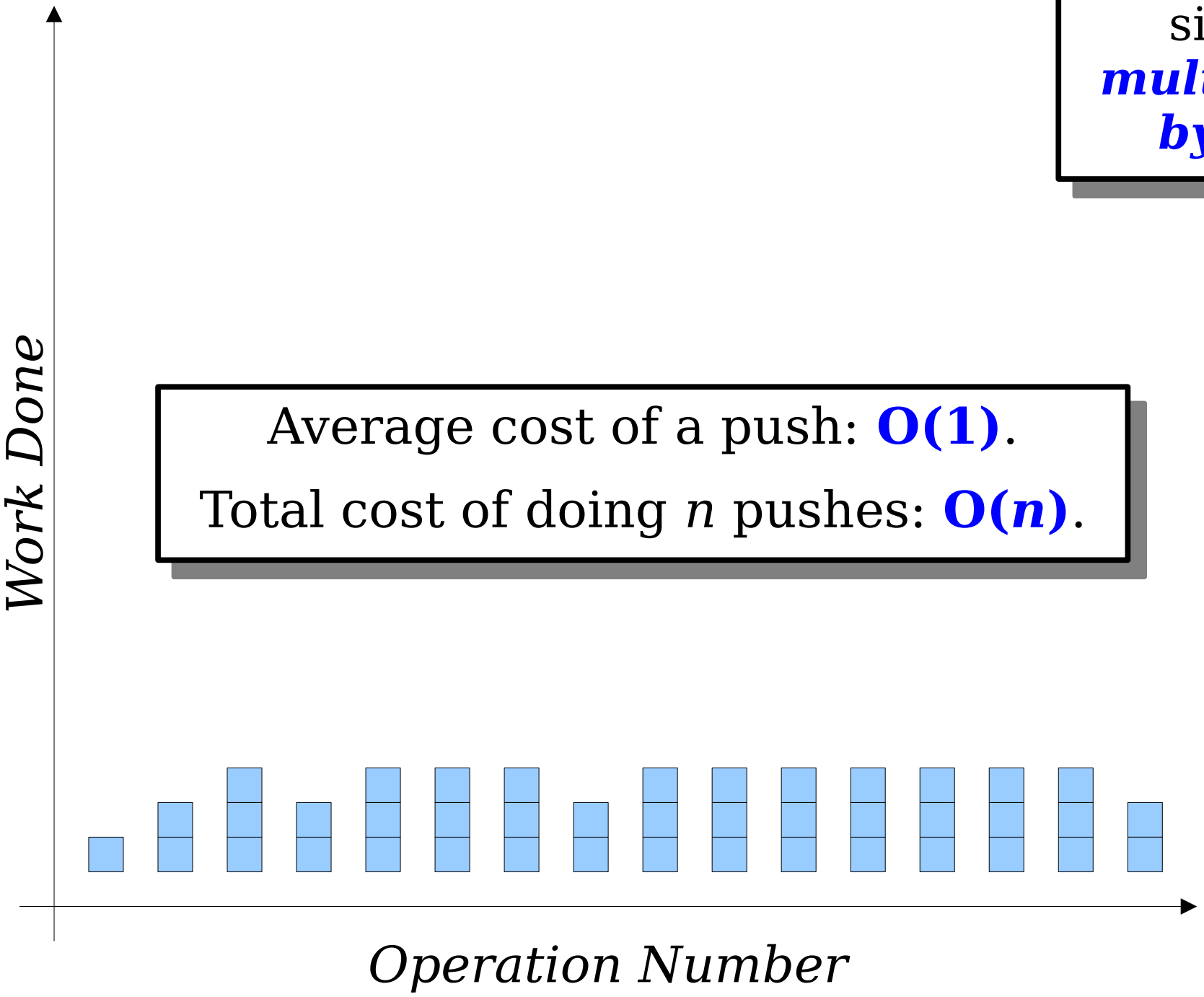
Increase array size by ***multiplying by two***.



Increase array size by ***multiplying by two***.

Average cost of a push: **O(1)**.

Total cost of doing *n* pushes: **O(*n*)**.



Amortized Analysis

- The analysis we have just done is called an ***amortized analysis***.
- We reason about the total work done by allowing ourselves to backcharge work to previous operations, then look at the “average” amount of work done per operation.
- In an amortized sense, our implementation of the stack is extremely fast!
- This is one of the most common approaches to implementing `Stack` (and `Vector`, for that matter).

Summary for Today

- We can make our stack grow by creating new arrays any time we run out of space.
- Growing that array by one extra slot or two extra slots uses little memory, but makes pushes expensive (average cost O(n)).
- Doubling the size of the array when we run out of space uses more memory, but makes pushes cheap (amortized cost O(1)).
- In practice, it’s worth paying this slight space cost for a marked improvement in runtime.

Your Action Items

- ***Read Chapter 11 and Chapter 12.1***
  - There's a lot of useful information there about dynamic memory allocation and class design.
- ***Start Assignment 5.***
  - Slow and steady progress is the name of the game here.
  - Ask for help if you need it! That's what we're here for.

Next Time

- ***No Class Monday***
- ***Then, When We Get Back…***
  - ***Hash Functions***
    - A magical and wonderful gift from the world of mathematics.
  - ***Hash Tables***
    - How do we implement `Map` and `Set`?