Lab 0: C Warmup

This lab does not count towards your total lab grade. Rather, it will help you better prepare for Lab 1 and **if you complete it for full credit you'll get an extra late day**.

You will also receive comments on your code style.

The goal of this lab is to prepare for Lab 1 by…

- Recalling how to program in C and use `valgrind`
- Practice with strings
- Simulate `send`/`recv` buffer management behavior

Part 1: Haiku – Practice with strings

Given the template below:

```
a new NOUN arrives
  the ADJECTIVE air begins to blow
    NOUN will come next
```

take user input for `NOUN`, `ADJECTIVE`, `NOUN` and fill it in.

Functions to implement in: `haiku.c`
Let's take a look at the starter code.

Read `man` pages

- A fun part of this call is you will learn new system calls and C functions. **You will need to get practice reading man pages** and figuring things out on your own (with a little bit of my help).
- You'll need to read about `strcat` and `sprintf`
- Some things to look for:
  - Example usage
  - What are the inputs and outputs?
  - What are the assumptions the function makes?
  - What happens when there is an error?
  - And when you're working on future labs: Is this a good tool for the job?

Always run your programs with `valgrind`

Bugs in these labs will often be challenging to debug. Even I have trouble sometimes helping students find really weird bugs. Get in the habit of using `valgrind` and `gdb`!

Example `haiku` program output

```
$ valgrind ./haiku
==2538227== Memcheck, a memory error detector
==2538227== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2538227== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==2538227== Command: ./haiku
==2538227==
Enter a noun: fall
Enter an adjective: cool
Enter a noun: homework
strcat version:
a new fall arrives
  the cool air begins to blow
    homework will come next
```

Part 2: Filling an integer array – Simulate socket buffer management

When you receive data from a network, you often don't know how much data the other side is going to send. To simulate, this you will write this function:

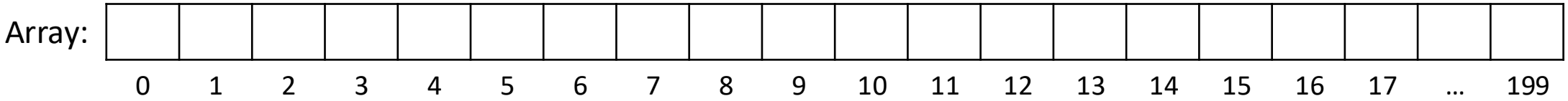
```
int get_ints(int *destination, int at_most);
```

You will repeatedly call this function to add numbers to an array (`destination`) until the function returns `0`, which means the generator is done creating numbers.

Part 2: Filling an integer array – Simulate socket buffer management

Files to implement: `fill_int_array.c`

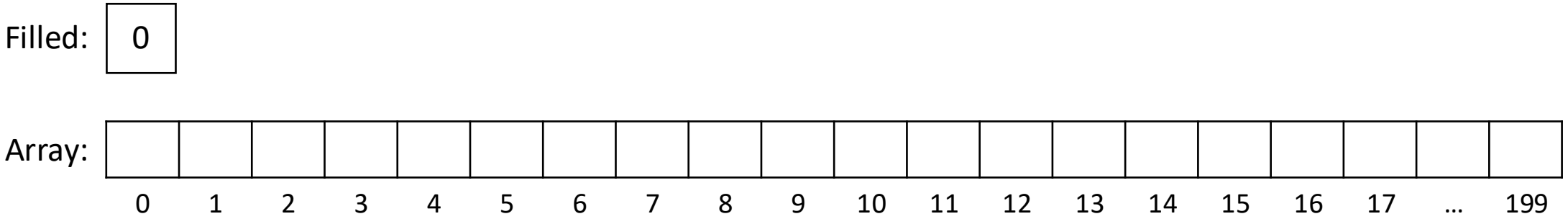
Let’s take a look at the starter code.

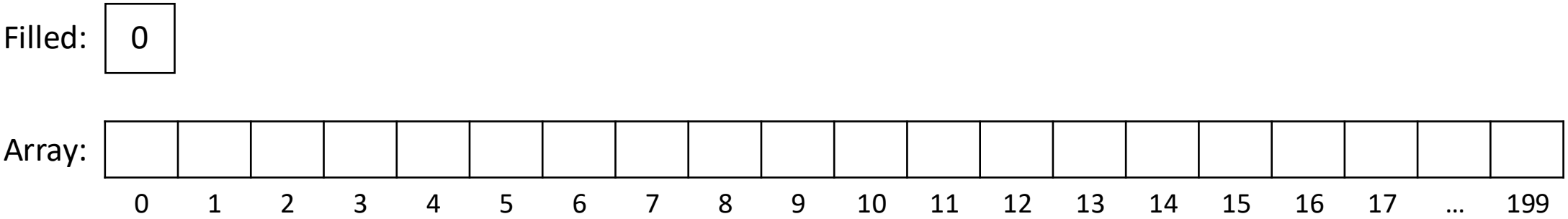


In the starter code `get_ints.h` there is a line:

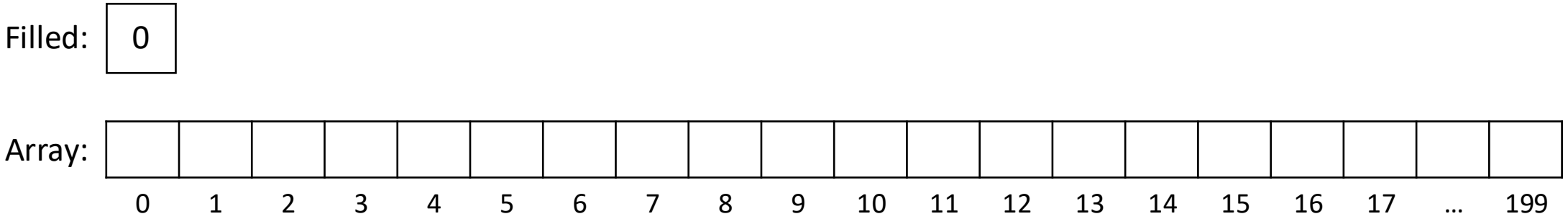
```
#define GET_INT_MAX (200)
```

that sets the maximum size of the destination array.

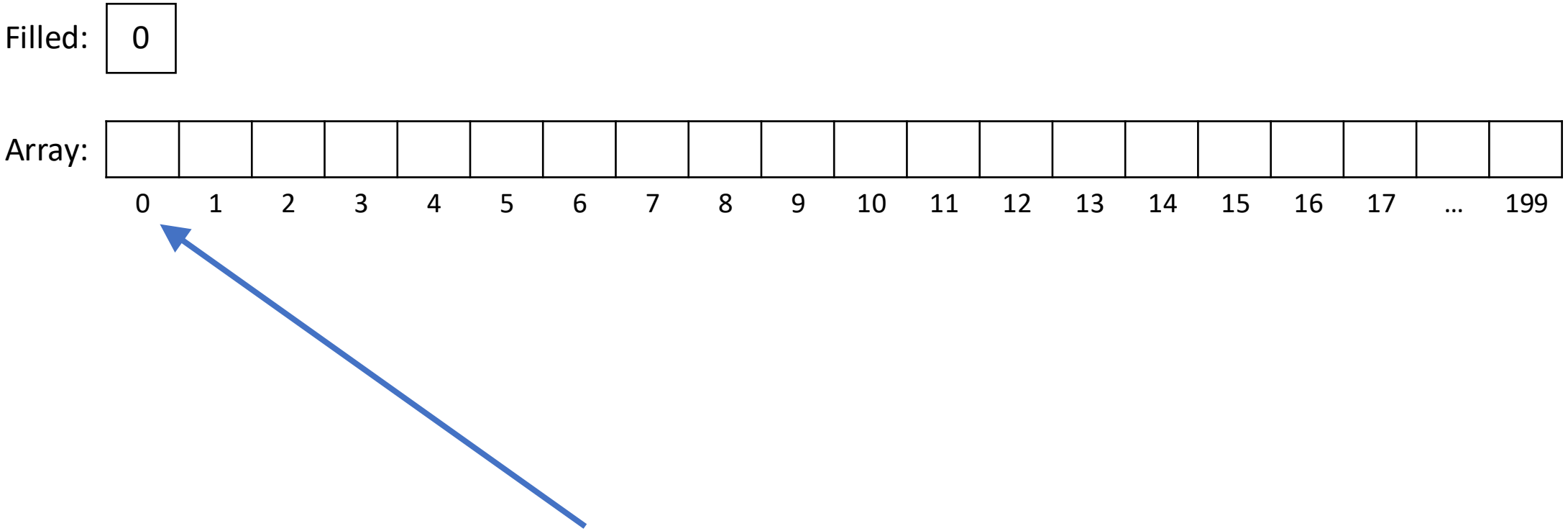

Filled:
0
Array:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 … 199



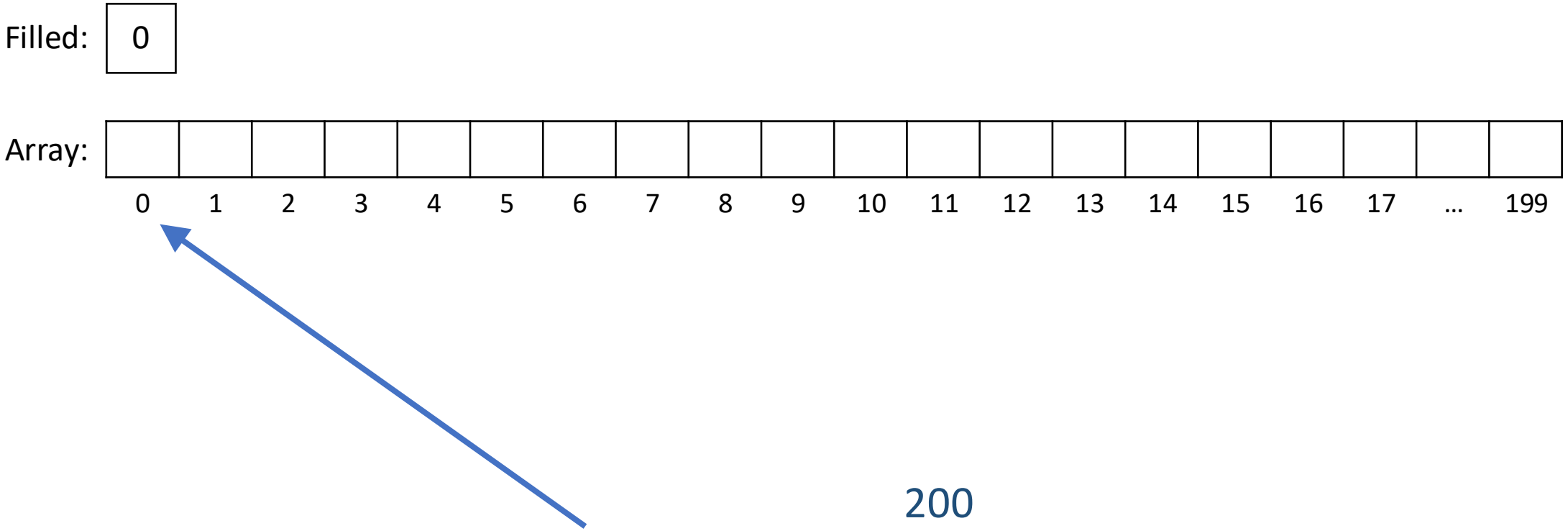
Call `setup()` to initialize the RNG.




```
int get_ints(int *destination, int at_most);
```




```
int get_ints(int *destination, int at_most);
```

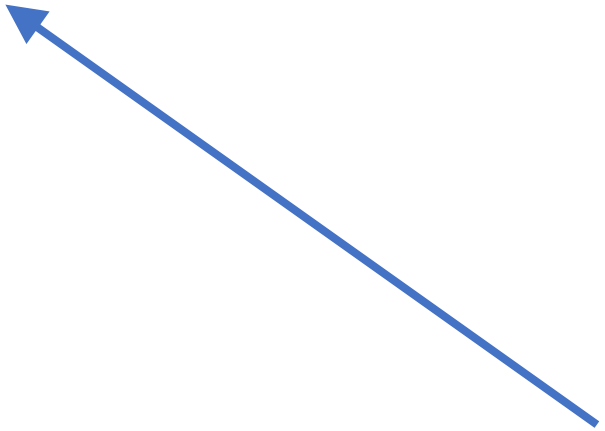



```
int get_ints(int *destination, int at_most);
```

Filled: 0

Array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	…	199



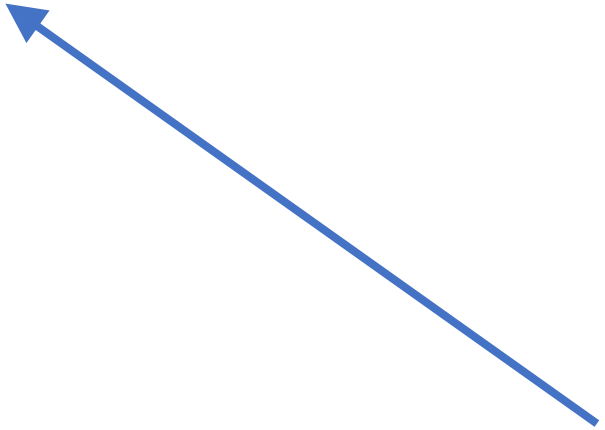
200

```
int get_ints(int *destination, int at_most);
```

Filled: 0

Array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	…	199



200

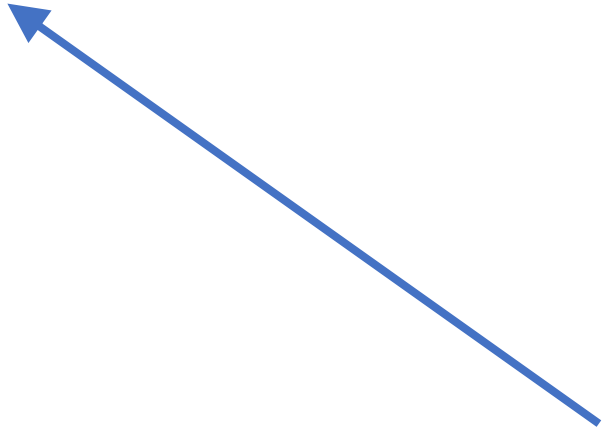
```
int get_ints(int *destination, int at_most);
```

4

Filled: 0

Array:

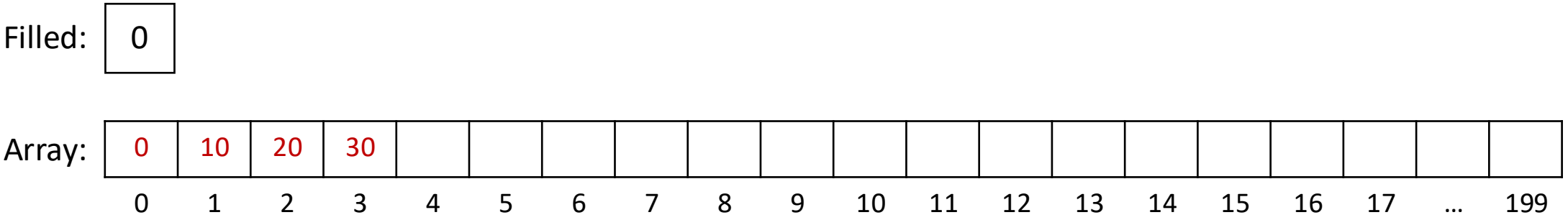
0	10	20	30																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	…	199



200

```
int get_ints(int *destination, int at_most);
```

4




```
int get_ints(int *destination, int at_most);
```

Returns 4

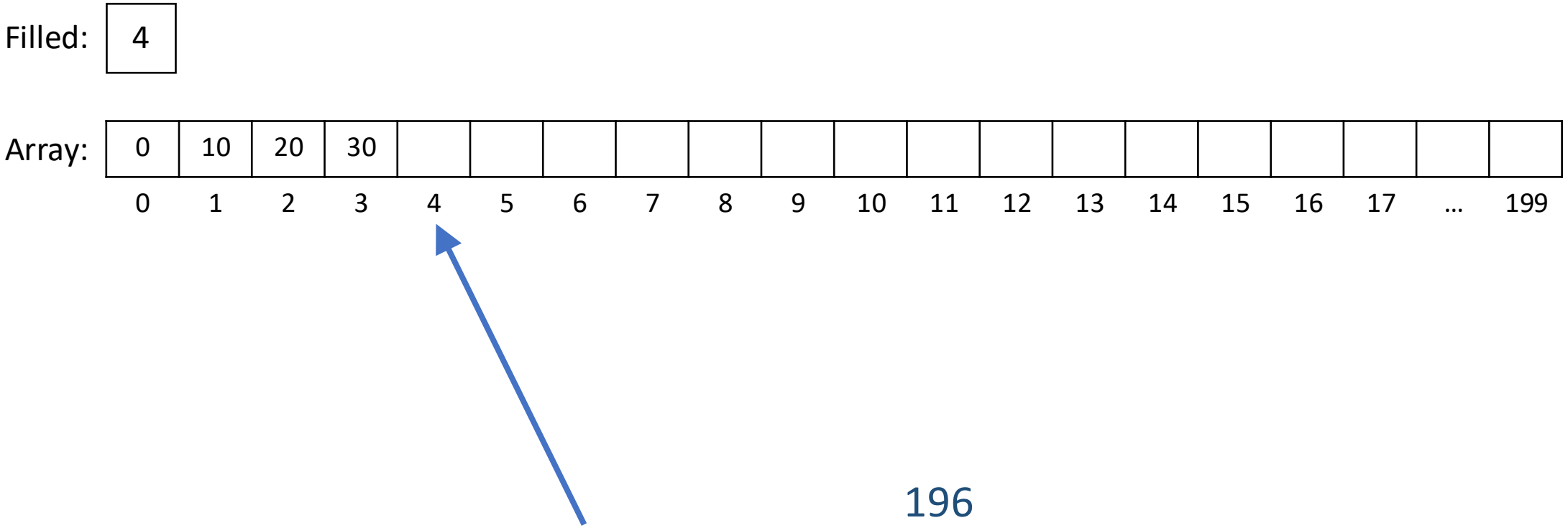
Filled: 4

Array:

0	10	20	30																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	…	199

```
int get_ints(int *destination, int at_most);
```

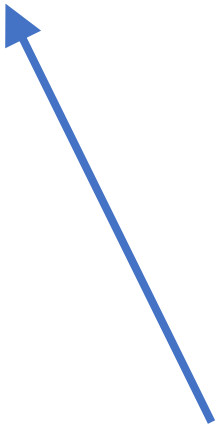
Returns 4




```
int get_ints(int *destination, int at_most);
```

Filled: 4

Array:	0	10	20	30	40	50	60	70	80	90										
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	…	199



196

```
int get_ints(int *destination, int at_most);
```

6

Filled: 4

Array:

0	10	20	30	40	50	60	70	80	90										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	…	199

```
int get_ints(int *destination, int at_most);
```

6

Filled: 10

Array:

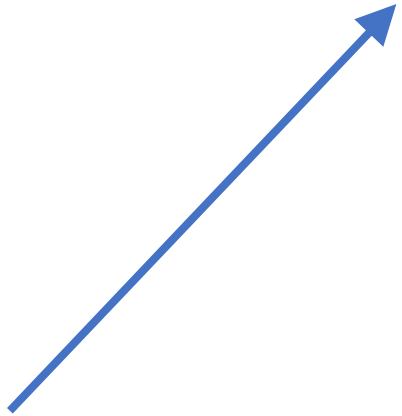
0	10	20	30	40	50	60	70	80	90										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	…	199

```
int get_ints(int *destination, int at_most);
```

6

Filled: 10

Array:	0	10	20	30	40	50	60	70	80	90										
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	…	199



190

```
int get_ints(int *destination, int at_most);
```

```
valgrind ./fill_int_array
==2549853== Memcheck, a memory error detector
==2549853== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2549853== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==2549853== Command: ./fill_int_array
==2549853==
Looks good!
==2549853==
==2549853== HEAP SUMMARY:
==2549853==     in use at exit: 0 bytes in 0 blocks
==2549853==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==2549853==
==2549853== All heap blocks were freed -- no leaks are possible
==2549853==
==2549853== For lists of detected and suppressed errors, rerun with: -s
==2549853== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```