COMP1511
Static arrays
Week 3 Lecture 2

functions/procedures recap

- Reusable blocks of code
- Callable multiple times
- variables within a function are scoped to that function

PI function

Would be annoying to write this every time we need to calculate!

```
double pi() {
  double sum = 0.0;
  for (int i = 0; i < 1000; i++) {
    sum += (-1.0) * pow(1.0 / 2.0, i) / (i + 1);
  }
  return 4.0 * sum;
}
```

Forward declaration

```
int main(void) {
    double calculated_pi = pi();
}

double pi() {
  double sum = 0.0;
  for (int i = 0; i < 1000; i++) {
    sum += (-1.0) * pow(1.0 / 2.0, i) / (i + 1);
  }
  return 4.0 * sum;
}
```

^ problem! `main` doesn't know that pi exists yet!

Forward declaration

```c
double pi();

int main(void) {
    double calculated_pi = pi();
}

double pi() {
  double sum = 0.0;
  for (int i = 0; i < 1000; i++) {
    sum += (-1.0) * pow(1.0 / 2.0, i) / (i + 1);
  }
  return 4.0 * sum;
}
```

^ Solved! We forward declared pi!

Quick functions recap demo

Arrays

So far, we can store a single item in each variable

What if you wanted to store many values?

Number of ice creams eaten

```
int day_1 = 2;
int day_2 = 3;
int day_3 = 3;
int day_4 = 5;
int day_5 = 7;
int day_6 = 1;
int day_7 = 3;
// Any day with 3 or more scoops is too much!
if (day_1 >= 3){
    printf("Too much ice cream\n");
}
if (day_2 >= 3) {...
```

Seem repetitive?

- Many variables would clutter the program
- Many variables would not always be efficient

Data structures

- Are common structures (not structs) used to store multiples of data
  - Usually (especially in COMP1511) of the same data type
- Can scale, easily storing a handful, up to thousands, or more elements of data!

Data structures in COMP1511

We will look primarily at two data structures:

- arrays (today)
- linked lists (future)

  These are very, very powerful data structures you will use forever

Arrays

- A collection of data, all of the same type. (homogonous)
- We have a single identifier for the entire array
- It is a random access data structure, meaning we can access any element in the array at any time

Arrays

- We can ready or modify individual elements
- It is a ***contiguous*** data structure

contigu-what?
Let's visualise arrays

Static arrays have a set size

(which you specify)

index:	0	1	2	3	4
values:					

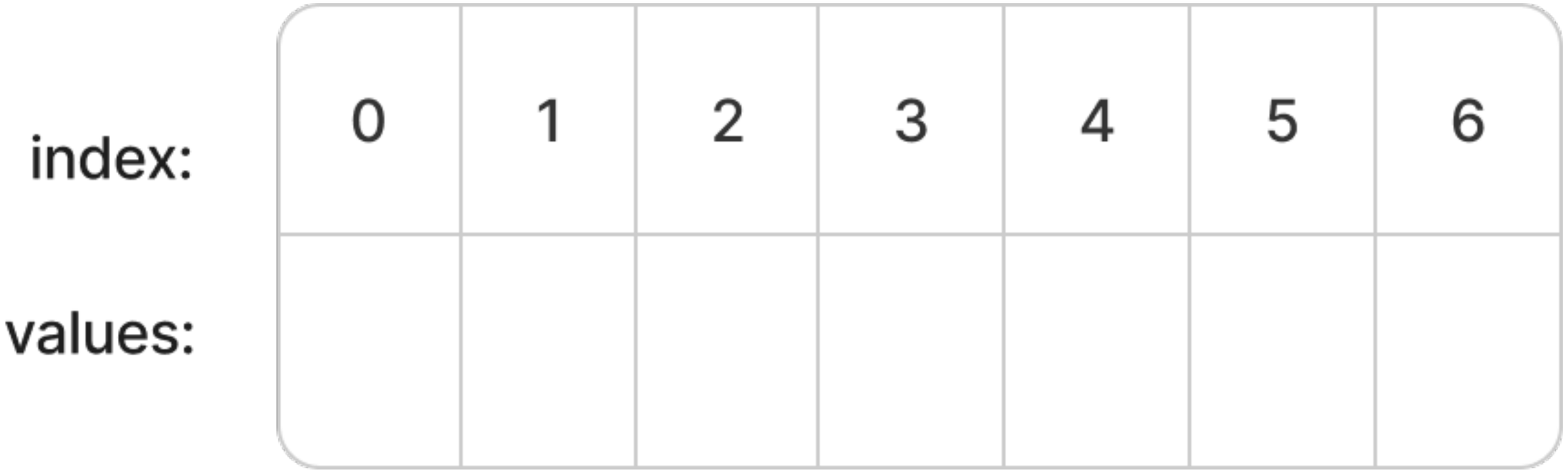
int array

index:	0	1	2	3	4
values:					

- This int array will store 5 integers
- 32bit * 5 elements = 160 bits of memory used

The array declaration syntax

```
int ice_cream_per_day[7];
```



Declare + initialise

```
int ice_cream_per_day[7] = {3, 2, 1, 2, 1, 3,
5};
```

^ Note you can only do this when you declare, not later!

```
int ice_cream_per_day[7] = {};
```

^ Will initialise all elements to 0

```
int ice_cream_per_day[7] = {3, 2, 1, 2,
1, 3, 5};
```

Creates:

index:	0	1	2	3	4	5	6
values:	3	2	1	2	1	3	5

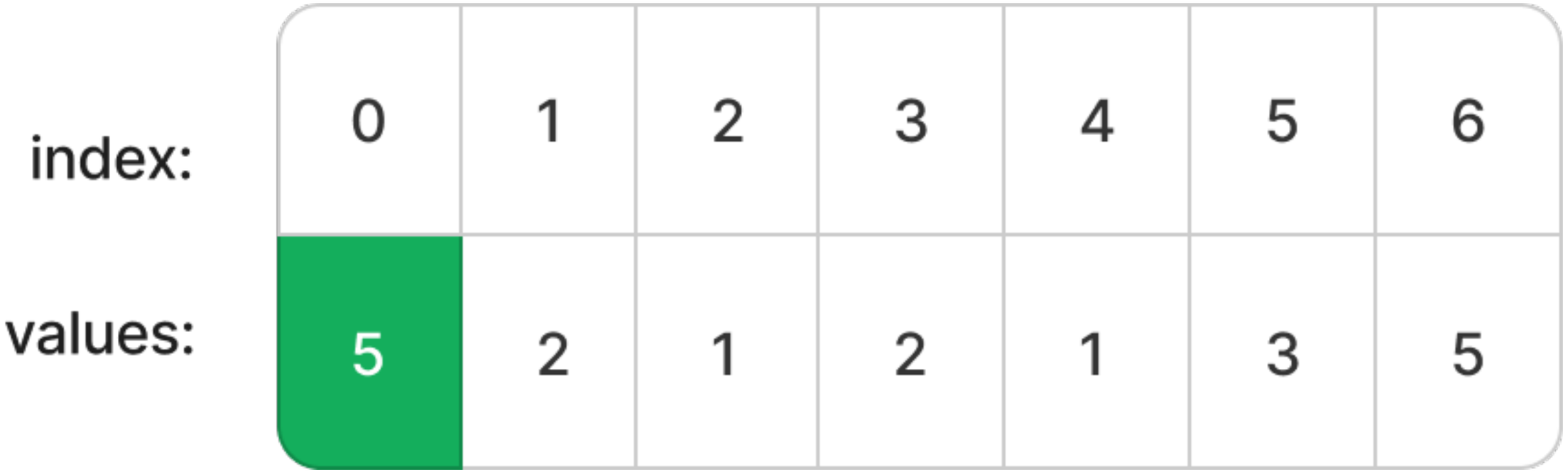
Accessing elements

```
int first_day_ice_creams =
ice_cream_per_day[0];
```

index:	0	1	2	3	4	5	6
values:	3	2	1	2	1	3	5

Writing elements

```
ice_cream_per_day[0] = 5;
```



arrays 🤝 loops

The power of arrays

```
int ice_cream_per_day[7] = {3, 2, 1, 2, 1, 3, 5};

// read each element
ice_cream_per_day[0];
ice_cream_per_day[1];
ice_cream_per_day[2];
ice_cream_per_day[3];
ice_cream_per_day[4];
ice_cream_per_day[5];
ice_cream_per_day[6];
```

^ Does this look repetitive?

If only we had a way to count :(

Bad

```
int ice_cream_per_day[7] = {3, 2, 1,
2, 1, 3, 5};

// read each element
printf("%d\n", ice_cream_per_day[0]);
printf("%d\n", ice_cream_per_day[1]);
printf("%d\n", ice_cream_per_day[2]);
printf("%d\n", ice_cream_per_day[3]);
printf("%d\n", ice_cream_per_day[4]);
printf("%d\n", ice_cream_per_day[5]);
printf("%d\n", ice_cream_per_day[6]);
```

Good

```
int ice_cream_per_day[7] = {3, 2, 1,
2, 1, 3, 5};

int i = 0;
while (i < 7) {
    printf("%d\n",
ice_cream_per_day[i]);
    i++; // i = i + 1;
}
```

Demo

Feedback

https://forms.office.com/r/K3PjvWebtD