13. Introduction to arrays

Herman Kamper

This lecture starts a new topic: arrays. An array is a list of things. That is all it is, and everything in this lecture follows from that one sentence, so if the middle gets murky, come back to it.

I want to say up front why this matters, because arrays look modest and are not. Almost all of the data that real programs work with is stored in lists. If you could open up the code behind a large language model, almost all of what that model actually *is* would be numbers sitting in arrays (these are called the model's parameters). Arrays are the basic building block, and once you have them you start seeing them everywhere.

Eight flats and eight variables

Here is a small and slightly silly problem that makes the point.

Say you own a block of flats. There are eight flats in it, and you want to keep track of how much water each one used this month. With only what we have covered so far, the program looks like this:

```c
#include <stdio.h>

int main()
{
    int flat1WaterLevel;
    int flat2WaterLevel;
    int flat3WaterLevel;
    int flat4WaterLevel;
    int flat5WaterLevel;
    int flat6WaterLevel;
    int flat7WaterLevel;
    int flat8WaterLevel;

    flat1WaterLevel = 2;
    flat2WaterLevel = 4;
    flat3WaterLevel = 6;
    flat4WaterLevel = 8;
    flat5WaterLevel = 10;
    flat6WaterLevel = 12;
```

```
    flat7WaterLevel = 14;
    flat8WaterLevel = 16;

    printf("Water levels of flat 1: %d kl\n", flat1WaterLevel);
    printf("Water levels of flat 2: %d kl\n", flat2WaterLevel);
    printf("Water levels of flat 3: %d kl\n", flat3WaterLevel);
    printf("Water levels of flat 4: %d kl\n", flat4WaterLevel);
    printf("Water levels of flat 5: %d kl\n", flat5WaterLevel);
    printf("Water levels of flat 6: %d kl\n", flat6WaterLevel);
    printf("Water levels of flat 7: %d kl\n", flat7WaterLevel);
    printf("Water levels of flat 8: %d kl\n", flat8WaterLevel);

    return 0;
}
```

Does it work? Yes. Is it nice? It is horrific. It does not even fit on a screen. It is tedious to type, tedious to read, and every one of those lines is an opportunity to make a typo that the compiler will happily accept.

Now suppose you do well for yourself and build two more flats on the side. You have to go back and add two more variables, two more assignments and two more `printf` calls, and you have to get all six of them right. The program does not scale.

What we want instead is a way to say: I am going to create a list of eight things, all at once; I am going to refer to the whole list by one name; and I am going to get at the individual items by their position in the list. That is exactly what an array is.

Defining an array

Defining an array looks a lot like defining a variable. The general form is:

```
type name[size];
```

You give the type of the things in the list, then a name, then in square brackets how many of them there are. For our flats:

```
int waterLevels[8];
```

That says: create a list of integers, call the list `waterLevels`, and make room for eight of them.

It helps to line this up against a variable declaration you already know:

```
// type   // name   // value
   int       a       = 2;

// type   // name   // size
   int       c       [ 5 ];
```

Both start with a type and a name. The difference is that the second one carries a size instead of a value. (We will see in a bit how we can put values inside the array elements.)

To see what that actually buys you, it is worth drawing what happens in memory when each of these two lines runs.

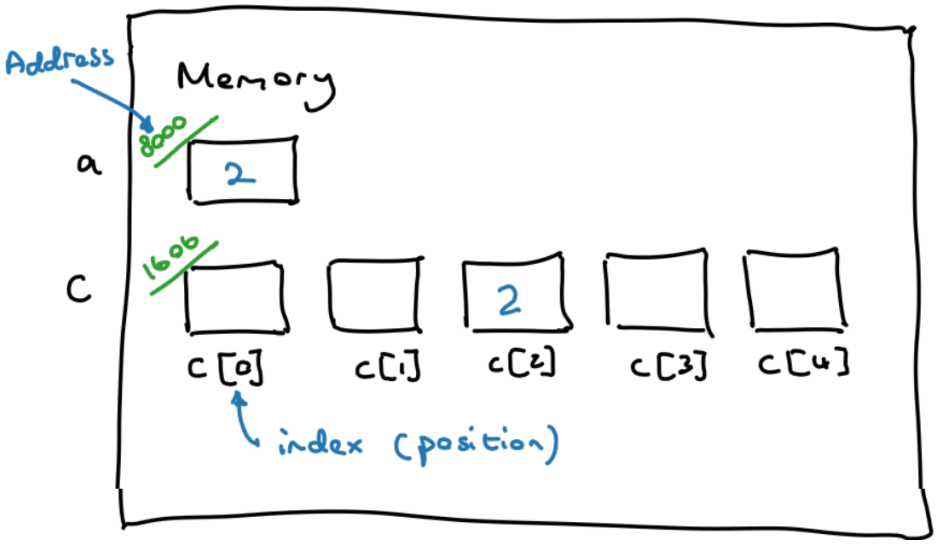

Figure 1: What the two declarations do in memory. `int a = 2;` creates a single box; `int c[5];` creates five consecutive boxes, each with its own name.

When `int a = 2;` runs, a box gets created somewhere in memory. The box is called `a`, it has a 2 inside it, and it sits at some specific address, say 8000. You have seen this a hundred times: a variable has a type, a name, a value and an address.

When `int c[5];` runs, C creates not one box but five. They all belong to `c`, they are all of type `int`, and because they are all the same type they sit right next to each other in memory: one block, then the next block, then the next. The whole run of them starts at some address, say 1606. Nothing has been put in the boxes yet, so at this point they contain rubbish.

The five boxes all belong to `c`, but each one has its own name. This box is `c[0]`, the next is `c[1]`, then `c[2]`, `c[3]` and `c[4]`. The number in the square brackets is the index, or the position: it says how far along the list you are.

And this is the part that makes arrays easy. Anywhere in your code, `c[2]` behaves exactly like an ordinary variable. It is a box. You can put things in it and take things out of it. Writing

```
c[2] = 2;
```

puts a 2 inside that particular box, in exactly the way that `a = 2;` puts a 2 inside `a`.

Engineers count from zero

Notice how the boxes were numbered above. When you create an array with size 5, the elements are 0, 1, 2, 3, 4. There is no fifth element. C allocates memory for five boxes, but it counts them starting from zero, so the last one is number 4.

This trips people up constantly, and it will trip you up too, so it is worth saying plainly: the size and the last index are different numbers. An array of size 5 has a last index of 4. An array of size 8 has a last index of 7. Engineers count from zero.

The name of the array

One more thing, and then you know almost everything you need to know about arrays.

The name `c`, on its own, is not one of the boxes. It refers to the whole run of them. But `c` does have an address, and that address is the address of the first element. In other words, the address of the array is the same as the address of `c[0]`.

That sounds like a technicality, and right now it is. It becomes important the moment we start handing arrays to functions.

Array elements are ordinary variables

Everything you can do with a variable, you can do with an element of an array. Assign to it:

```
c[0] = 3;
```

Read from it, and combine several of them in an expression:

```
printf("%d", c[0] + c[1] + c[2]);
```

Use one in a calculation and store the answer somewhere else:

```
x = c[6] / 2;
```

And here is the one that catches people. The thing inside the square brackets does not have to be a number you typed. It can be any expression that produces an integer. Suppose `a` is 3 and `b` is 2. What does this line do?

```
c[a + b] += 2;
```

Work it from the inside out. First `a + b` is evaluated, giving 5. So the line is the same as `c[5] += 2;`. That opens box number 5, looks at what is inside, adds 2 to it, and puts the result back in the box.

Box number 5 is the sixth element of the array, because the first one is `c[0]`. If that made you hesitate, good: it is exactly the confusion that zero-based counting causes, and the only cure is to count it out a few times until it stops feeling strange.

Back to the water levels

Now we can throw away that horrible program and write the version we actually wanted. In this building the ground floor uses the least water and each floor above uses a bit more, so flat 1 used 2 kl, flat 2 used 4 kl, flat 3 used 6 kl, and so on.

```c
/*
 * Declaring and using arrays.
 * Author: Herman Kamper
 */

#include <stdio.h>

int main()
{
    int iFlat;
    int waterLevels[8];

    for (iFlat = 0; iFlat < 8; iFlat++)
    {
        waterLevels[iFlat] = (iFlat + 1)*2;
    }

    for (iFlat = 0; iFlat < 8; iFlat++)
    {
        printf(
            "Water levels of flat %d: %d kl\n",
            iFlat + 1, waterLevels[iFlat]
        );
    }

    return 0;
}
```

Twenty-six lines of tedium have become two short loops, and it prints:

```
Water levels of flat 1: 2 kl
Water levels of flat 2: 4 kl
Water levels of flat 3: 6 kl
Water levels of flat 4: 8 kl
Water levels of flat 5: 10 kl
Water levels of flat 6: 12 kl
Water levels of flat 7: 14 kl
Water levels of flat 8: 16 kl
```

This is the point where the array and the loop have to click together, and in my experience it is the single place in this lecture where people get lost. So let us go through the first loop one line

at a time.

`int waterLevels[8];` creates eight boxes in memory. They all belong to `waterLevels`, and their names are `waterLevels[0]` through `waterLevels[7]`.

`iFlat = 0` creates a variable `iFlat` somewhere in memory and puts a 0 inside it. Then the loop checks its condition: is 0 less than 8? It is, so we go in.

Inside, `waterLevels[iFlat] = (iFlat + 1)*2;` is evaluated. `iFlat` is 0, so the right-hand side is `(0 + 1)*2`, which is 2. And the left-hand side is `waterLevels[0]`, the first box. So a 2 gets put in the first box.

We reach the bottom of the loop body, `iFlat++` runs, and `iFlat` goes from 0 to 1. Is 1 less than 8? It is, so back in we go. Now the right-hand side is `(1 + 1)*2`, which is 4, and the left-hand side is `waterLevels[1]`, the second box. A 4 goes in the second box.

Round again: `iFlat` becomes 2, `waterLevels[2]` gets 6. The memory now looks like the figure below.

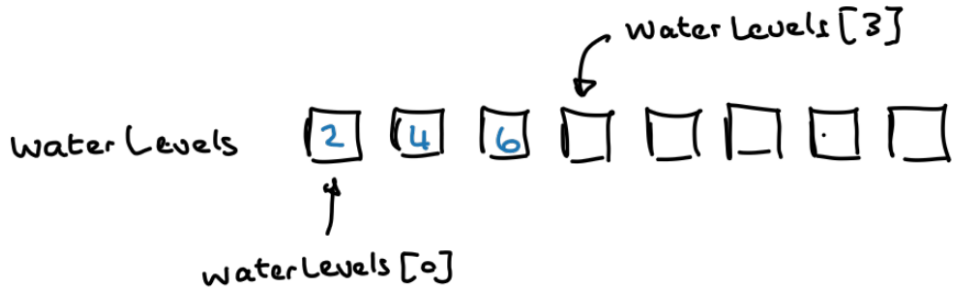

Figure 2: A visualisation of the computer's memory as we are filling up the `waterLevels` array.

The loop continues until `iFlat` becomes 8. Now the condition `8 < 8` is false, and the loop stops.

The second loop is the easy one. It does not calculate anything. It just opens box 0, box 1, box 2, and so on, and prints what it finds.

Two small things in that program are worth pointing out.

The condition is `iFlat < 8`, not `iFlat <= 8`. With <= the loop would run one extra time with `iFlat` equal to 8, and it would try to use `waterLevels[8]`, which does not exist. This is such a common mistake that it has a name – an off-by-one error – and it is worth building the habit of checking that condition every single time you write a loop over an array.

The other is the `iFlat + 1` inside the `printf`. The array counts from 0, but the tenants of the building do not: nobody lives in flat 0. So the loop counts from zero and the output adds one back on. Remove the `+ 1` and the program will happily tell you about the water usage in flat 0 and flat 7.

Initialising arrays

You are used to declaring a variable and giving it a starting value in one go, with `int a = 2;`. You can do the same with arrays, and there are three ways to do it.

The first is to list the values you want:

```
int n[5] = {1, 2, 3, 4, 5};
```

The 1 goes in `n[0]`, the 2 in `n[1]`, the 3 in `n[2]`, the 4 in `n[3]` and the 5 in `n[4]`.

The second is for when you just want everything set to zero, which turns out to be extremely common:

```
int n[5] = {0};
```

This creates five boxes and puts a 0 in every one of them. The rule that makes it work is that if the initialiser list is shorter than the array, the remaining elements are set to 0. So you supply one zero and C fills in the other four.

That rule is worth reading twice, because it is not "repeat the value I gave you". If you write `int n[5] = {8};` you do not get five eights. You get 8, 0, 0, 0, 0. The `{0}` trick works only because the value you want everywhere happens to be the same value C pads with.

The third way is the lazy version of the first. Leave the size out and let C count for you:

```
int n[] = {1, 2, 3, 4, 5};
```

C counts the initialisers, finds five of them, and makes a five-element array.

One thing that is *not* on that list: there is no automatic initialisation. Just like ordinary variables, the elements of `int c[5];` are not set to anything. Whatever happened to be in that memory is what you will read back.

The `#define` directive

There is one more command to meet, and it is a slightly odd one.

Look back at the water-level program and count how many times the number 8 appears: once in the declaration, once in the first loop condition, once in the second. Three places, in a program that is fifteen lines long. In a real program it might be thirty places. Change the building to twenty flats, miss one of them, and you have a bug that will not announce itself.

The fix is `#define`. It goes right at the top of your program, above `main`:

```c
/*
 * Declaring and using arrays.
 * Author: Herman Kamper
 */

#include <stdio.h>

#define FLATS 8

int main()
{
    int iFlat;
    int waterLevels[FLATS];

    printf("No. flats: %d\n", FLATS);

    for (iFlat = 0; iFlat < FLATS; iFlat++)
    {
        waterLevels[iFlat] = (iFlat + 1)*2;
    }

    for (iFlat = 0; iFlat < FLATS; iFlat++)
    {
        printf("Water levels of flat %d: %d kl\n", iFlat + 1, waterLevels[iFlat]);
    }

    return 0;
}
```

Now the size lives in exactly one place. Want twenty flats? Change the 8 to a 20 and you are suddenly a much wealthier person.

What `#define` actually does is worth being precise about, because it is not really a C command at all. It is a preprocessor directive. Before your program gets compiled, a separate step called the preprocessor reads through the file, finds every occurrence of `FLATS`, and replaces it with `8`. Only then does the compiler see the code, and what it sees is the original program with the number typed out three times. It is a find and replace, run automatically, just before compilation.

That last point matters. A student asked whether you can think of `FLATS` as a kind of global variable. Not really, and the difference is the useful part: a global variable creates a box in memory that exists while the program runs. `#define` creates nothing. There is no box, no memory, no variable. The text of your program is edited before it is compiled, and that is the whole story.

By convention these names are written in capitals, which is a habit worth keeping. If you write `#define flats 8` and then use `FLATS` in the code, nothing gets replaced, and the error you get will point at the wrong place entirely.

A longer example: the student food survey

Let us do something more substantial with this.

> Ten students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 5, where 1 means awful and 5 means excellent. Place the responses in an integer array and summarise the results of the poll.

That is the problem. Note that it is not fully specified: what exactly does "summarise" mean? Before writing a single line of code, we need to answer that, and we should answer it away from the keyboard.

Step one: work out what you want

Let us make up some responses. Say the ten students gave these ratings, in order:

`1, 1, 1, 2, 1, 3, 4, 1, 1, 4`

Now, what do we want on the screen at the end? Something like this: how many students gave 1 star? How many gave 2 stars? And so on up to 5. Count them by hand from the list above before reading on.

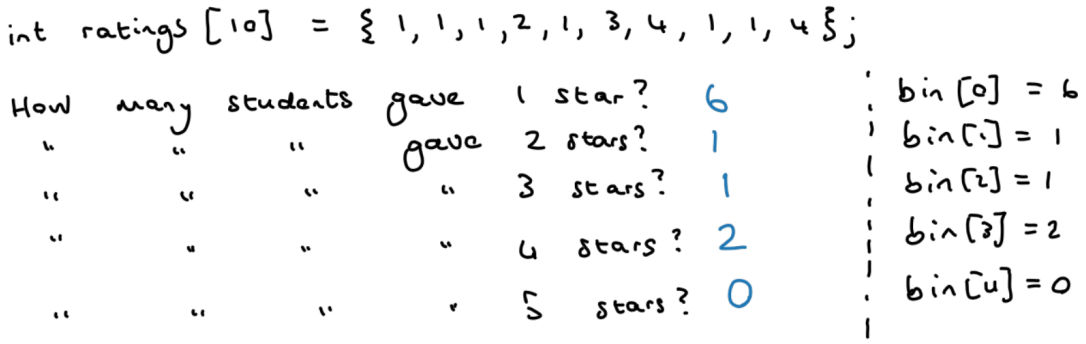

Figure 3: Working the problem out before writing any code: the made-up ratings, the questions we want answered, and the counts that answer them. The counts on the right are written as `bin`; below we call that array `ratingBins`.

Six students gave 1 star. One gave 2 stars. One gave 3. Two gave 4. Nobody gave 5, which is a sad outcome for the cafeteria.

Those five counts are the summary. If you drew them as bars you would have a histogram, and the cafeteria would like a tall bar on the right-hand side. We are not going to draw it; we are just going to print the numbers.

The word that goes with histograms is bins. Each response gets dropped into a bin according to what it was: all the 1s go in one bin, all the 2s go in another. What we want at the end is how much is in each bin.

So we need a second array, which we will call `ratingBins`, holding the counts we just worked out:

```
ratingBins[0] = 6
ratingBins[1] = 1
ratingBins[2] = 1
ratingBins[3] = 2
ratingBins[4] = 0
```

Take a moment on the fact that there are two arrays here and they are different sizes and mean different things. `ratings` has one element per student, so it has ten elements, and each holds a number between 1 and 5. `ratingBins` has one element per possible rating, so it has five elements, and each holds a count between 0 and 10. Getting these two confused is the main way this program goes wrong.

We have not written any code. But we now know exactly what the program has to do, which is the hard part.

Step two: write it

Start with the two arrays:

```c
#include <stdio.h>

#define NO_OF_STUDENTS 10    // number of ratings
#define POSSIBLE_RATINGS 5   // 1 to 5

int main()
{
    int i, rating;

    // Declare and initialise array with ratings from students
    int ratings[NO_OF_STUDENTS] = {1, 1, 1, 2, 1, 3, 4, 1, 1, 4};

    // Declare and initialise bin array, containing the counts for the
    // different ratings
    int ratingBins[POSSIBLE_RATINGS] = {0};
```

The ratings are hard-coded here to keep things simple. The bins are initialised to all zeros using the `{0}` trick from earlier, which is exactly right: before anyone has voted, every count is zero.

How many bins are there? Five, not four. The ratings run from 1 to 5, so there are five possible answers, and they will live in bins 0 through 4.

Now loop over the students:

```c
    // For each response by a student
    for (i = 0; i < NO_OF_STUDENTS; i++)
    {
        // Determine the rating
        rating = ratings[i];
```

At this point the loop does nothing useful; it just walks through the students and pulls out each rating into a variable. Strictly we do not need that variable, because we could use `ratings[i]` directly everywhere. But the next line is the one that fries brains, and having a plainly named `rating` sitting there makes it much easier to think about.

Here is that next line:

```
        // Increment the element in the count/bin array
        ratingBins[rating - 1] += 1;
    }
```

Why the minus one? Because the ratings run from 1 to 5 and the bins are numbered 0 to 4. A student who gave 1 star belongs in bin 0. A student who gave 4 stars belongs in bin 3. Subtracting one converts from the rating scale that humans used to the index scale that C uses.

Now let us run that line by hand, because it does more than it looks like it does.

Before the loop starts, the five bins all contain 0. The first student gave a 1. So `rating` is 1, and `rating - 1` is 0. We go to box number 0. We take out what is inside it, which is 0. We add 1 to that, giving 1. And we put that result back in the box. Bin 0 now contains 1.

The second student also gave a 1, so bin 0 goes from 1 to 2. The third student gave a 1: bin 0 becomes 3. The fourth student gave a 2, so this time `rating - 1` is 1, we open bin 1, find a 0 in it, and it becomes 1. The fifth student gave a 1, taking bin 0 to 4. And on it goes.

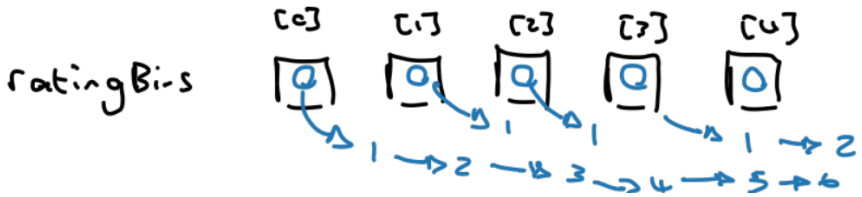

Figure 4: The five bins filling up as each response is processed. Every arrow is one student's rating being counted.

By the end, bin 0 holds 6, bin 1 holds 1, bin 2 holds 1, bin 3 holds 2 and bin 4 holds 0, which is precisely the summary we worked out on paper before we started.

The thing to hold on to is that

```
ratingBins[rating - 1] += 1;
```

is nothing exotic. It is exactly `a = a + 1`, where `a` happens to be one of the boxes in an array rather than a standalone variable. Open the box, take out the number, add one, put it back. If it helps, write it the long way and convince yourself they are identical:

```
ratingBins[rating - 1] = ratingBins[rating - 1] + 1;
```

All that is left is to print the results:

```
    // Print ratings
    for (i = 0; i < POSSIBLE_RATINGS; i++)
    {
        printf("%d students gave %d stars\n", ratingBins[i], i + 1);
    }

    return 0;
}
```

And there is the `+ 1` again, for the same reason as in the water-level program. We loop over the bins from 0 to 4, but we print the star ratings as 1 to 5, because that is what the students were actually asked.

Putting it all together:

```
/*
 * Summarise food ratings.
 * Author: Herman Kamper
 */

#include <stdio.h>

#define NO_OF_STUDENTS 10    // number of ratings
#define POSSIBLE_RATINGS 5   // 1 to 5

int main()
{
    int i, rating;

    // Declare and initialise array with ratings from students
    int ratings[NO_OF_STUDENTS] = {1, 1, 1, 2, 1, 3, 4, 1, 1, 4};

    // Declare and initialise bin array, containing the counts for the
    // different ratings
    int ratingBins[POSSIBLE_RATINGS] = {0};

    // For each response by a student
    for (i = 0; i < NO_OF_STUDENTS; i++)
    {
        // Determine the rating
        rating = ratings[i];

        // Increment the element in the count/bin array
        ratingBins[rating - 1] += 1;
    }
```

```
    // Print ratings
    for (i = 0; i < POSSIBLE_RATINGS; i++)
    {
        printf("%d students gave %d stars\n", ratingBins[i], i + 1);
    }

    return 0;
}
```

which prints:

```
6 students gave 1 stars
1 students gave 2 stars
1 students gave 3 stars
2 students gave 4 stars
0 students gave 5 stars
```

Notice how little of the work was typing. Nearly all of it happened before the editor was open, when we decided what the two arrays were and what the bins meant. Once that was settled, the program was about fifteen lines. This is the normal shape of programming, and it is worth getting used to.

Notice too what the program does *not* care about. There is nothing in it that depends on there being exactly ten students. Change `NO_OF_STUDENTS` to 40, put forty ratings in the list, and every line still works.

Looking forward

Arrays give you a way to hold a whole collection of related values under one name and to reach any of them by position. Combined with a `for` loop, that means you can write one short piece of code that handles ten items or ten thousand.

Everything in this lecture has kept the array inside `main`. The obvious next question is what happens when you want to hand one to a function, and the answer turns out to be more interesting than you would expect, because of that small remark earlier about the array's name being the address of its first element. That is where the next lecture starts, along with how to search through an array for something.

Exercises

1. Take the water-level program and, without running it, write down what happens if you change the loop condition from `iFlat < 8` to `iFlat <= 8`. Then run it and see. Explain what you observe.
2. The food survey hard-codes the ten ratings. Modify the program so that it reads each rating from the user with `scanf` instead. You will need a loop and `&ratings[i]` (why?).
3. Write a program that stores ten integers in an array and calculates their sum, then their average. Be careful about the type of the average.
4. Write a program that fills an array with the first twenty even numbers, and then prints them in reverse order.
5. Predict what `int n[5] = {7};` puts in each of the five elements. Write your prediction down, then check it. Now do the same for `int n[5] = {7, 7};`.