14. Strings, arrays and functions, searching arrays

Herman Kamper

The last lecture introduced arrays: list-like variables where a whole row of boxes shares one name, and you open a particular box by writing its index in square brackets. This lecture takes that one idea and pushes it in three directions. First, strings, which turn out to be nothing more than arrays of letters. Second, what happens when you hand an array to a function, which is stranger than you would expect and is the most important thing here. Third, searching: given an array and a value, how do you find where that value sits?

Let me start with the reason strings matter. The best website on earth is Wikipedia. It is, more or less, a summary of all human-produced knowledge, and it is worth noting that they do not allow AI-generated content on it, which I agree with: the AI has to get its knowledge from somewhere, and Wikipedia is that somewhere.

Figure 1: The most important website on earth, and every last bit of it is text. Image from Wikimedia Commons.

Everything on Wikipedia is text: letters stitched together into words and sentences. So if we want to write programs that do anything with the world's knowledge, we need a way to hold letters, and specifically combinations of letters, inside a variable. That is what strings give us.

Strings are just arrays

You already know about arrays, so you almost already know about strings. There are only a handful of special things to learn.

Start with a single letter. We have had this since early on:

```
char character = 'X';
```

In memory that creates one box, the box is called `character`, and inside it sits the single letter `X`. Note the single quotes: single quotes mean one character.

A string is just a sequence of characters, so a string is an array of characters. And you already know how to make an array: give it a type, give it a name, put square brackets after the name, and initialise it.

```
char stringA[] = {'s', 't', 'a', 'r', '\0'};
```

That line creates an array called `stringA`. How many boxes? Five. If I had more self-respect I would have written the 5 in the brackets myself, but I was lazy, so C counted the initialiser list for me and put the 5 there. Inside the five boxes it places `s`, `t`, `a`, `r` and then that odd-looking `'\0'`.

That last one is the first special thing about strings. `'\0'` is called the null character, and every string in C ends with it. It is a convention, but it is a convention the whole language relies on, and in a moment you will see exactly what it buys us.

The second special thing is that the creators of C decided that writing out every letter with its own quotes was too much work. So for character arrays specifically, they gave us a shorthand:

```
char stringB[] = "wars";
```

Double quotes. This creates exactly the same kind of thing as the line above: an array of characters. So single quotes give you one character, double quotes give you an array of them.

Now, how many boxes does `stringB` have? It looks like four, and this is the thing everyone forgets: it is five. When you use double quotes, C automatically adds one extra box at the end and puts `'\0'` inside it for you. That is the whole convenience of the shorthand, and it is also the part that trips people up when they start counting.

You can also make a character array without saying what goes in it:

```
char stringC[20];
```

That is twenty boxes, and because it is uninitialised there is nothing meaningful in any of them.

A question that comes up here: since we did not write the size for `stringA` and `stringB`, is the size somehow flexible afterwards? No. It is never fluid in C. What happens is that at compile time C counts what is in the braces, or counts the letters in the quotes and adds one for the null character, works out the size once, and fixes it. After that the array is that size, permanently.

Below is a figure of what all of this looks like in memory. The bottom row is `stringC` after the user has typed something into it, which we get to shortly.

Printing a string is done with `%s`, and printing one character out of it is done with `%c`:

```
printf("String is %s %s\n", stringA, stringB);
printf("The 2nd character is %c\n", stringB[1]);
```

The second line prints `a`, not `w`. `stringB[1]` opens up the box at index 1, and we count from zero, so that is the second box.

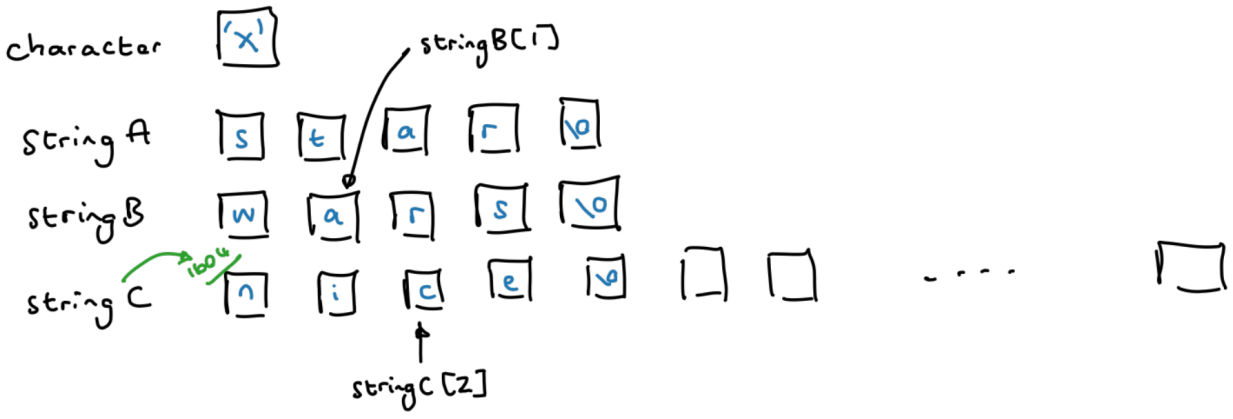

Figure 2: Three character arrays in memory. `stringB[1]` opens up one specific box and gives you the character `a`. The name `stringC` on its own is the address where the array starts, marked in green.

Reading a string from the user

You can probably guess the function: `scanf`, with `%s`.

```
char stringC[20];

printf("Enter a new string: ");
scanf("%s", stringC);
printf("You entered: %s\n", stringC);
```

Look hard at that `scanf` line, because something is missing. Where is the ampersand? Every other time we have used `scanf` we have written `&` in front of the variable name.

The answer is the one loose end from the last lecture, and it is worth understanding properly rather than memorising. What does `stringC` mean, exactly? `stringC[0]`, `stringC[1]` and `stringC[2]` each mean a specific box, and each of those behaves like an ordinary variable. But `stringC` on its own, with no brackets, does not mean the contents of anything. It means the address where the array starts: the green 1604 in the figure above.

For an ordinary variable, `scanf` needs to be told where the box lives rather than what is in it, which is why we write `&`. But an array's name is already an address. So writing `stringC` gives `scanf` exactly what it wants, and adding `&` would be saying "the address of the address".

This is the important bit, so I will say it once more plainly: write the name of an array without an index, and what you get is the address of the array.

If you are wondering whether `&stringC` would break: annoyingly, it usually appears to work. The reason is that the address of the array and the address of its first box are the same number, so `scanf` ends up writing to the right place by accident. The types are not the same, though, and if you compile with warnings on, the compiler tells you so. Do not write it. Code that only works because two different things happen to have the same numeric value is code that will eventually stop working.

The rule flips as soon as you put brackets back on. `stringC[2]` is just a character, an ordinary single variable like any other, so if you want to read into it you are back to needing the address:

```
scanf("%c", &stringC[2]);
```

Two more things about reading strings. A space ends the string as far as `scanf` is concerned: type `hello there` and `stringC` gets `hello`, with the null character straight after it. And an array of twenty boxes holds at most nineteen characters, because the twentieth is needed for `'\0'`. If you type more than that, you are writing into memory that is not yours. On my machine the program died with a message reading `stack smashing detected`, which sounds much cooler than it is.

So when the user types `nice` into `stringC`, five of the twenty boxes get used: `n`, `i`, `c`, `e`, `'\0'`. The remaining fifteen still contain whatever junk was there before. That is fine and completely normal. The string ends where the null character says it ends, and nobody looks past it.

Walking through a string

Now the null character earns its keep. Here is a loop that prints a string with a space between every character, so `nice` comes out as `n i c e`:

```
for (i = 0; stringC[i] != '\0'; i++)
{
    printf("%c ", stringC[i]);
}
```

Read the loop condition. It does not say `i < 20`. It does not mention the length of the string at all. It says: keep going as long as the character you are looking at is not the null character. You will see this shape of loop constantly in code that works with strings, and this is why every string ends in `'\0'`: it means a piece of code can walk through a string without being told how long it is.

Step through it once with `nice` in `stringC`. Start at `i` equal to 0. Is `stringC[0]` the null character? No, it is `n`, so print `n` and a space. Increment `i` to 1. Is `stringC[1]` the null character? No, it is `i`, so print that. Same for `c` at index 2 and `e` at index 3. Now `i` is 4, and `stringC[4]` is `'\0'`, so the condition is false and the loop ends. Four characters printed, and the fifteen boxes of junk after them never get touched.

Putting the whole thing together:

```c
#include <stdio.h>

int main()
{
    int i;
    char character = 'X';

    // Define strings in two ways
    char stringA[] = {'s', 't', 'a', 'r', '\0'};
    char stringB[] = "wars";
    char stringC[20];

    // Print strings
    printf("String is %s %s\n", stringA, stringB);
    printf("The 2nd character is %c\n", stringB[1]);

    // Read a string from scanf and print it
    printf("Enter a new string: ");
    scanf("%s", stringC);
    printf("You entered: %s\n", stringC);

    // Print the last string, but add a space between
    // each character
    for (i = 0; stringC[i] != '\0'; i++)
    {
        printf("%c ", stringC[i]);
    }

    return 0;
}
```

which, if you type `nice`, prints:

```
String is star wars
The 2nd character is a
Enter a new string: nice
You entered: nice
n i c e
```

One last remark before we move on. There is nothing magic about the name `stringC`. Call it `apples` if you like; it is the `char` and the brackets that make it a string, not the name. And notice that none of this is really new machinery. It is arrays, with letters in the boxes, plus the `'\0'` convention. That is the whole summary of strings.

Passing arrays to functions

Change of topic, and this is the part that is worth your full attention. A great deal of what real programs do is pass arrays around. Most of the programs I have written spend their time handing arrays of neural network weights from one function to another. So: how do you give an array to a function?

Here is a small program that does two things side by side, one with an ordinary variable and one with an array.

```
#include <stdio.h>

void incrementInt(int x);
void incrementFirstElement(int x[]);

int main()
{
    int i = 5;
    int array[] = {4, 0, 6};

    incrementInt(i);
    printf("Value of i: %d\n", i);

    incrementFirstElement(array);
    printf("Value of array[0]: %d\n", array[0]);

    return 0;
}

void incrementInt(int x)
{
    x++;
    printf("Value of x inside function: %d\n", x);
}

void incrementFirstElement(int x[])
{
    printf("Current value of x[0]: %d\n", x[0]);
    x[0] = x[0] + 1;
    printf("Current value of x[0]: %d\n", x[0]);
}
```

Look at how the second function is declared. To tell a function it will be receiving an array, you write the type, the parameter name, and then empty square brackets: `int x[]`. Notice what is not in the brackets: the size. You do not put it there, and the function has no way of working it out. Hold onto that thought, because we come back to it.

To call the function you write the array's name with no brackets at all:

```
incrementFirstElement(array);
```

Now predict the output before reading on. In `incrementInt`, `x` starts at 5, gets incremented to 6, and 6 is printed inside the function. Then we return to `main` and print `i`. In `incrementFirstElement`, `x[0]` starts at 4, gets incremented to 5, and both are printed. Then we return to `main` and print `array[0]`. Here is what actually comes out:

```
Value of x inside function: 6
Value of i: 5
Current value of x[0]: 4
Current value of x[0]: 5
Value of array[0]: 5
```

Most people get the first four lines right and the last one wrong. The ordinary variable behaved as it always has: the function changed `x` to 6, but back in `main` the value of `i` is still 5. The function got a copy. But the array did not behave that way at all. The function changed `x[0]` to 5, and back in `main`, `array[0]` is now 5 as well. The original was modified.

This picture is the whole explanation.

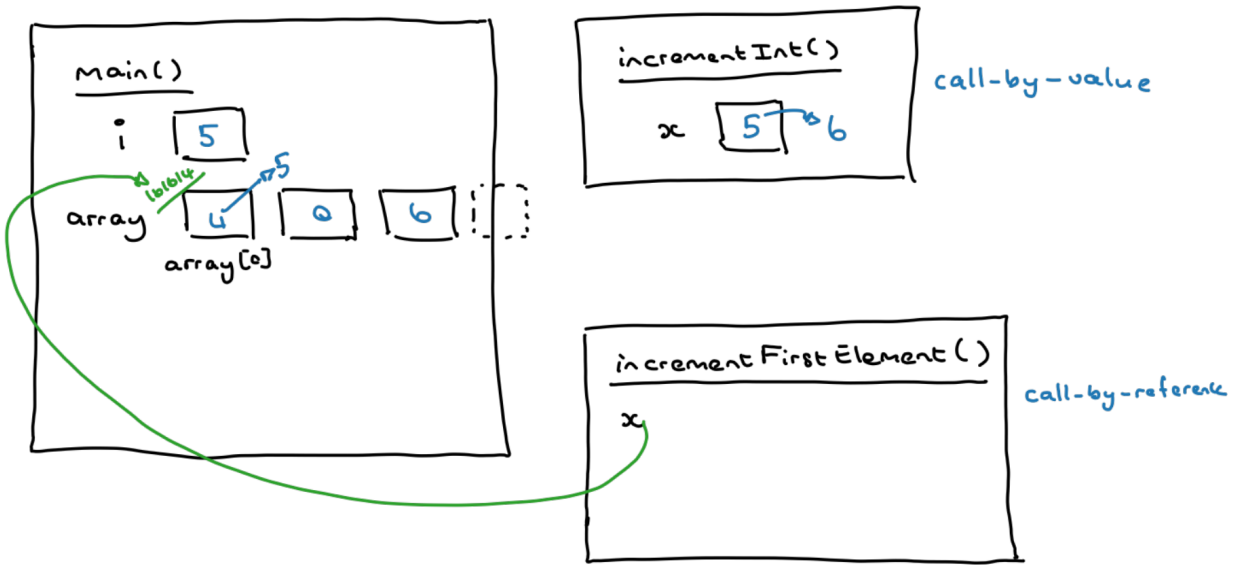

Figure 3: What happens when you pass an ordinary variable versus an array. `incrementInt` gets its own box with a copy of 5 in it. `incrementFirstElement` gets no boxes at all, just the address 161614, which points straight back at the original array.

Work through the left-hand side first. In `main` there is a box called `i` with 5 in it, and next to it three boxes holding 4, 0 and 6, called `array[0]`, `array[1]` and `array[2]`.

When we call `incrementInt(i)`, C opens up the box called `i`, takes out the value 5, and copies it into a brand new box called `x` that belongs to the function. There are now two boxes with a 5

in them. Incrementing `x` changes the function's box and leaves `main`'s box alone. This is called call-by-value, and it is everything we have done with functions so far.

Now the right-hand side. When we call `incrementFirstElement(array)`, what does `array` mean on its own, without brackets? We settled this earlier: it is the address. So what gets copied into the function is not the values 4, 0 and 6. It is the number 161614.

This means the function does not have three boxes of its own. It has no boxes at all, just an address, and its `x` points back at the original array in `main`. When the function says `x[0]`, what really happens is: go to that address, add zero, and look at whatever is in that box, which is 4. If it said `x[1]` it would go to the address, step forward by one integer, and find 0. And when the function runs `x[0] = x[0] + 1`, it goes to that address and changes the 4 into a 5 – and that box is `main`'s box. There is no copy to change. This is called call-by-reference.

So the important thing to walk away with is this: if you pass an array to a function and the function changes it, the original array changes. That is often exactly what you want, and it is occasionally a disaster.

There is one exception, and it follows from the same rule rather than fighting it. If you pass a single element rather than the whole array:

```
myFunction(array[3]);
```

then the square brackets open up that one box and pull the value out, and a value is what gets passed. So an individual array element is passed by value, like any ordinary variable, and the function cannot change the original. Brackets, copy. No brackets, address.

Two consequences worth knowing

The first consequence is that the function does not know how big the array is. All it received was a starting address. Nothing in `int x[]` says three, or twenty, or a million. So the standard thing to do, and you will see this everywhere, is to pass the size along as a second argument:

```
void myFunction(int b[], int arraySize);
```

This matters more than it sounds, because C will not stop you walking off the end. Try printing `array[3]` for our three-element array. There is no error, no complaint, no crash. It prints whatever happens to be sitting in the next patch of memory. Running it three times in a row on my machine gave 158998016, then 1880823040, then 1780953088. Keeping track of how far you are allowed to go inside an array is your job and nobody else's.

The second consequence is that sometimes you want to hand an array to a function and be certain it will not be tampered with. C has a keyword for that:

```
void myFunction(const int b[], int arraySize);
```

Declare the parameter `const` and the function is not allowed to modify the array. Any attempt to write to it is a compile error. This is genuinely useful once your program grows into many functions and you want to be sure that none of them can quietly wreck data belonging to another.

An example: searching arrays

On to the last topic, which is something you end up doing constantly: given an array full of values, find where a particular value sits.

Say I have a list of marks and I want to find the first person who scored exactly 74, so I can go and look at their script and see whether there is one more mark hiding in it somewhere. I do not want to know whether 74 is in the list. I want to know its index, so I can go and do something with it. The value being hunted for has a name: it is called the key.

The obvious approach could not be simpler: start at the beginning, look at every element in turn, and stop when you find the key. That is called a linear search.

Here is the setup. These are ratings that students gave me, and one of them is a 9. I would like to know which one.

```
int lecturerRatings[6] = {99, 75, 9, 65, 70, 55};
int searchKey = 9;
int studentIndex;

studentIndex = linearSearch(lecturerRatings, searchKey, 6);
printf("Student no %d gave 9%%\n", studentIndex);
```

Notice that the call passes three things: the array by name, the key we are after, and the size, because as we established the function cannot work the size out for itself.

Now the function. Let me build it in pieces rather than writing it down whole. Start with a loop that just walks the indices and shows what is in each box, which is a good habit whenever you are unsure whether a loop is doing what you think:

```
int linearSearch(int array[], int key, int size)
{
    int index = 0;

    while (index < size)
    {
        printf("array[%d] = %d\n", index, array[index]);
        index++;
    }

    return index;
}
```

That prints all six elements and returns 6. Now add the actual searching. The moment the element under `index` equals the key, we are done and want out of the loop:

```
int linearSearch(int array[], int key, int size)
{
    int index = 0;

    while (index < size)
    {
        if (array[index] == key)
        {
            break;
        }
        index++;
    }

    return index;
}
```

Trace it. `index` starts at 0. Is 0 less than 6? Yes. Is `array[0]`, which is 99, equal to 9? No, so `index` becomes 1. Is `array[1]`, which is 75, equal to 9? No, so `index` becomes 2. Is `array[2]`, which is 9, equal to 9? Yes, so we break out, skip straight to the `return`, and hand back 2. The program prints:

```
Student no 2 gave 9%
```

which is the answer we wanted.

You can also fold the test into the loop condition and do away with the `break` entirely:

```
while ((index < size) && (array[index] != key))
{
    index++;
}
```

That is shorter, and it says the same thing: keep stepping as long as you have not run out of array and have not found the key. But the order of those two tests is not a matter of taste. `&&` evaluates left to right and stops as soon as it knows the answer, so with `index < size` written first, `array[index]` is never touched once `index` has reached `size`. Swap them around and the last thing the loop does before giving up is read one element past the end of the array. It will usually seem to work, which is the worst possible outcome, because it means you will not find out until much later.

One last improvement. As written, a failed search returns `size`, which the caller has to know to check for. It is cleaner to return an index when you find the key and a value that cannot possibly be an index when you do not:

```
int linearSearch(const int array[], int key, int size)
{
    int index;

    for (index = 0; index < size; index++)
    {
        if (array[index] == key)
        {
            return index;
        }
    }

    return -1;
}
```

Note the `const`: a search has no business modifying what it is searching, so we say so. Searching for 9 gives 2, and searching for 42 gives -1.

Linear search will work on any array at all, in any order, which is its great virtue. Its vice is the cost. If the value is not there, you have looked at every single element, and if the array has a million elements in it, that is a million comparisons.

Looking forward

Three things came together in this lecture, and they are more connected than they look. Strings turned out to be arrays with letters in them and a null character on the end. Passing an array to a function turned out to mean passing an address, which is why the function can change your original data and why it never knows the size. And searching turned out to be the first real algorithm we have written.

In the next lecture we will look at an even better search algorithm, binary search.

Exercises

1. Write a program that reads a word from the user into a `char` array and counts how many characters it has, without using any library function. Then check your answer against `strlen` from `string.h`.
2. Take the string program from this lecture and print the string backwards. You will need to find the end of the string first.
3. Write a function `void doubleAll(int a[], int size)` that doubles every element of an array. Call it from `main`, then print the array afterwards to confirm that the original really did change.
4. Now write `int sumOf(const int a[], int size)` which returns the sum without modifying anything. Try adding a line inside it that changes `a[0]` and see what the compiler says.
5. Predict what happens if you pass an array of 5 elements to a function but tell the function the size is 10. Then try it. Why does the compiler not catch this?
6. Write a linear search that returns the index of the *last* occurrence of the key rather than the first.