

Arrays or "numbered variables"

Comments and questions to [John Rowe](#).

In this lecture we are mainly studying arrays, which allow us to refer to "variables" by **number**, not just by name.

We will also take to opportunity to introduce some concepts that are particularly helpful when dealing with arrays. Symbolic constants allow us to give a name to a constant which we can then use in the rest of the program and "in-place" arithmetic operators give a convenient short-hand for expressions such as `j = j + 1;`.

Warm-up discussion

We are used to referring to things such as years, hours etc. by number. Turn to your neighbour(s) and discuss some or all the following:

- What was the first year of the twenty-first century?
- If this is the twenty-first century, why do the years start with "20"?
- Why is zero after nine on a phone or computer keyboard keyboard?
- Is "12 pm" twelve noon or twelve midnight?
- Most analogue locks only cover half the dial. If you were to design a twenty-four hour analogue clock, what numbers would you put on the dial?

When to use arrays

Coordinates, vectors and matrices

When referring to the position of a point in space we use three coordinates, e.g. (1.0, 1.0, 2.3). More generally, we refer to the components on an n -dimensional vector, \mathbf{V} , as V_i where i is an integer, and a *matrix*, M , is a two-dimensional array of numbers. M_{ij} .

C's **arrays** give us a convenient way of defining these and referring to their individual components, or elements, as `v[i], m[i][j]`, etc.

If set up an array called, say `myarray` of length twenty, we can then treat any of its elements, such as `myarray[4]`, `myarray[17]` just like a variable. More to the point, we can now use expressions such as `myarray[i]`, etc., which is why arrays are so useful.

It is usually better to use an array for coordinates than individual variables x, y and z .

Other uses

Imagine we are calculating the sides and angles of a triangle. We might give the variables for the lengths of the sides names like "a", "b" and "c". But what do we do if we are dealing with a decagon? The code will be full of repeated lines of code with slightly different variable names, "a", "b", ... "k".

Similarly, in a game with several players each with their own score it's better to have an array `score` rather than individual variables called `score1`, `score2`, etc.

Arrays help us to avoid "copy-and-paste loops" in much the same way as functions.

Declaring and using one-dimensional arrays

As we see above, C uses square brackets `[]` to designate the number in an array, both when declaring it and accessing the individual elements.

Array example 1: declaration and simple assignment

The following code sets up an array of three `doubles` and then sets the value of its elements.

```
int main() {
```

```

double sides[3];

sides[0] = 5.2;
sides[1] = 3.1;
sides[2] = 4.6;

... /* Do something interesting */
return 0;
}

```

The first statement ("double sides[3];") reserves enough space to store three values, in this case doubles. Arrays can have any type: int, long, etc.

The following lines set the values of the three individual elements. The number inside the [] is referred to as that element's **index** and can be any integer expression, not just a constant.

The first element of the array has index zero, the last has index N-1

As can be seen above, C numbers its arrays **from zero**. Although intuitively our reaction is usually to number things starting from "1", as the warm-up discussion shows, it tends to lead to problems later on. Therefore all modern numbering systems tend to start from zero. For example, the twenty-four clock numbers its hours from 0-23 (not 1-24) and its minutes and seconds from 0-59.

If we have an array of size N and the first element has index zero then the last element must have index N-1, **not N!**.

The classic array error

```

double sides[3];

sides[3] = 1.7; /* WRONG ! */

```

Be sure to remember this one!

This is a particularly nasty bug which we shall discuss slightly more later in the lecture. It is a special case of an invalid index value, **which C does not check for**. Trying to access the N+1th value of an array of size N is like standing three steps away from a cliff-top and taking four steps forward.

Array example 2

Here we use scanf() to read in sides[0] and sides[1] and use them in arithmetic expression, again just like a variable.

```

int main() {
    double sides[3];

    printf("Please enter the two sides of a right-angled triangle\n");
    scanf("%lf %lf", &sides[0], &sides[1]);

    sides[2] = sqrt(sides[0] * sides[0] + sides[1] * sides[1]);

    ... /* Do something interesting */
    return 0;
}

```

Array initialisation

The first example above sets the value of side's three elements manually. Whilst this is allowed, in general we prefer to initialise arrays and variables at the point of declaration to make it more difficult to forget to miss one out.

Reminder: initialising variables

Setting the value of variables using an ordinary assign statement makes it too easy to make a mistake:

```

double x, y, z;

/* Easy to miss one out without noticing */
x = 1.0;
y = 2.0;
z = 3.0;

```

Here we have correctly remembered to initialise all three variables just after declaration but it would be easy to have forgotten one.

The preferred method is to initialise as part of the declaration. It's also shorter and easier to understand:

```
/* Harder to miss one out without noticing */
double x = 1.0, y = 2.0, z = 3.0;
```

Initialising an array

Initialising the elements of an array is done using `{ ... }`:

```
double sides[3] = { 5.2, 3.1, 4.6 };
```

We have already seen `{ ... }` used to group several **statements** together, here they are used to group several **initialisers** together.

Two array initialisation short cuts

Array initialisation has a couple of nice features.

Automatic array sizing

If we miss out the array size the compiler infers the array size from the number of initialisers:

```
double sides[] = { 5.2, 3.1, 4.6 };
```

Missing initialisers are treated as zeros

Conversely, if we provide fewer initialisers than required the rest of the array elements are initialised to **zero**:

```
int myarray[3] = { 1 };
```

Here we have shown an array of integers for variety. `myarray[0]` is initialised to 1, the other two elements to zero (not one!!).

This does not apply to uninitialised arrays, only to arrays with fewer initialisers than values. To initialise an array to all zero use:

```
int anotherarray[3] = { 0 };
```

"In place" arithmetic operators

In the loops we have written so far we have often had steps such as:

```
i = i + 1;
j = j - 1;
factorial = factorial * n;
```

which have the same variable on both sides of the equals sign. Similarly, in the wages calculation in the preface we had a step:

- Add that day's pay to the running-total

This sort of phrase comes naturally to us and C provides a way of doing it directly with the `+=` operator:

```
running_total += days_pay;
```

which means, unsurprisingly, "add the value of `days_pay` to `running_total`". Notice there is no space between the `+` and the `=`.

Just about any operator can have "=" after it:

```
... = ... /* ... */
```

```

x += y;      /* add y to x */
x -= y;      /* subtract y from x */
x *= y;      /* multiply x by y */
x /= y;      /* divide x by y */

```

The code fragments:

```

j += 1;
k -= 1;

```

can be further abbreviated to:

```

++j;      /* add 1 to j */
--k;      /* subtract 1 from k */

```

These last two are used quite often (particularly ++j) so try to use them whenever possible.

Warning: j++ vs. ++j

There are two slightly different forms of the "++" operator. Both have the same effect when in a statement on their own but are different when used in an expression, such as an assignment.

The statement

```
k = ++j;
```

means "add one to j and then set k to the (new) value of j".

```
k = j++;
```

means "set k to the (old) value of j and then add one to j".

Similarly the two snippets:

```
int j = 5;
myarray[j++] = 11;
```

```
int j = 5;
yourarray[++j] = 11;
```

Set the values of `myarray[5]` and `yourarray[6]` to the value 11.

Exactly the same principle applies to the "--" operator.

I suggest you don't use "++" inside expressions. it just gives you the chance to go wrong.

Remember the mega-principle and avoid chances to go wrong!

Looping over an array

We can use the "++" operator together with a `while()` loop to loop over an entire array, in this case to read it in:

```

void loopdemo1() {
    int numbers[10];
    int i;

    i = 0;                                /* Initialise */
    while ( i < 10 ) {                    /* Test */
        printf("Please enter value %d\n", i);
        scanf("%d", &numbers[i]);
        ++i;                               /* Increment */
    }
    ... /* Do something interesting */
}

```

Whilst this loop is valid, there are two things we can do to make it clearer and simpler, and to reduce the chances of making mistakes.

1. Avoid "magic numbers", use symbolic constants instead.

In this code the size of the array (10) is hard-coded in two different places. This is sometimes referred to as a "magic number": it's just there and we have no idea why. If we were to change the size of the array to 12 we would have to change the limits of our loops to 12 as well. Worse, there may be other occurrences of the number 10 that we don't have to change, as they have the value 10 for a different reason. And if we see the number "20" we will need to ask if it should now have the value "24".

The following # (preprocessor) line allows us to give a name to a constant:

```
#define VALUES 10
```

- Any subsequent occurrence of the word `VALUES` (outside of text strings) is replaced by the rest of the line; in this case "10".
- Like all "#" directives, it is line-based with no semi-colon
- It's conventional to use ALL UPPER-CASE letters, and digits for symbolic constants.

We shall expect you to always use symbolic constants for array dimensions, except for applications where it is logically impossible to change the array size, for example when dealing with a triangle.

Remember: don't give yourself the opportunity to go wrong!

2. Simplify a common loop structure

The `while()` loop has the Initialise - Test - Increment pattern we have often seen before:

1. **Initialise:** `(i = 0;)`
2. Loop:
 - a. **Test** `(i < 10)` and if false quit the loop.
 - b. Execute the body.
 - c. **Increment.** `(++i;)`

with the loop continuing until the Test is false.

This pattern occurs so frequently that C has a special loop for it: the `for()` loop.

The for() loop

The `for()` loop has the following format:

```
for (Initialise; Test; Increment ) {
    ... /* Loop body */
}
```

Note that the three expressions are separated by **semicolons**, not commas, and that they occur in the same order as they are used. The steps in the `for()` loop are done in exactly the same order as in the equivalent `while()` loop:

1. **Initialise**
2. Loop:
 - a. **Test** and if false quit the loop.
 - b. Execute the body.
 - c. **Increment.**

Just as in the `while()` loop, **if the test is initially false the body never executes.**

Putting these two together we have the following archetypal "loop over an array" code to read in the masses of some stars:

```
#define NSTARS 10

void loopydemo() {
    double mass[NSTARS];
    int star;

    for (star = 0; star < NSTARS; ++star ) {
        printf("Please enter the mass of star %i\n", star);
        scanf("%lg", &mass[star]);
    }
    ... /* Do something interesting */
}
```

Notice how we have taken the opportunity to use a descriptive variable name for our array index.

This is the third and final of C's loops, joining the `while()` and `do ... while()` loops we have already met.

The classic array error, revisited

It's essential to use "less than" (<) in our for() loop, not "less than or equal to" (<=) as the latter will make us go one over the end of the array, in the same way as in classic array error we showed earlier. Here we try to read in one too many integers (where we assume VALUES was #defined earlier):

```
int numbers[VALUES], i;

for (i = 0; i <= VALUES; ++i) { /* Wrong ! */
    fscanf(infile, "%d", &numbers[i]);
}
```

Be sure to remember this one too!

Passing an array to a function

Having declared an array in one function, we sometimes want to pass that array to another function.

Example 1: printing an array using a function

```
/*
 * Demonstrate passing an array to a function
 */
#include <stdio.h>
#define VALUES 1000000

void printarray(int output[VALUES]);

int main() {
    int numbers[VALUES], i;

    /* Initialise to uninteresting values */
    for(i = 0; i < VALUES; ++i)
        numbers[i] = i;
    printarray(numbers);

    return 0;
}

void printarray(int output[VALUES]) {
    int i;
    FILE *outfile;

    if ((outfile = fopen("mydata.dat", "w")) == NULL) {
        fprintf(stderr, "Failed to open mydata.dat for output\n");
        return;
    }

    for (i = 0; i < VALUES; ++i) {
        fprintf(outfile, "%d\n", output[i]);
    }
}
```

How would we like C to pass the array numbers to the function printarray()?

One option would have been to make the name of the array "numbers" a synonym for the whole array and to make a copy of it to pass to printarray(). But that would be a huge task for an array of a million elements. Besides, experience shows that when we pass an array to a function we do sometimes want to change it - it could be quite handy to have a function called readarray() for example.

This is one reason why C adopts the following rule:

When an array name is passed to a function, what is passed is the location of the beginning of the array. [Kernighan and Ritchie].

This has two consequences:

1. The computer only has to pass a single address (typically four or eight bytes) to the function not the entire array, in this case a million elements.

Kernighan and Ritchie wrote the book ("K&R") that first defined the C language.

2. When a function modifies the array **it is modifying the original array.**

A slightly less obvious consequence is that passing the size of the array to a function doesn't really do anything. If we have declared an array with dimension 3, then even if we tell a function it has dimension 1000, it still really only has dimension 3. For this reason, in a list of array arguments the size of a one-dimensional array (and more generally, the left-most dimension of a multi-dimensional array) can be omitted.

Arrays and memory

The elements of a one-dimensional array are stored sequentially in memory

An array is one of the few times we can be sure of the order of two things in the computer's memory.

For example, if we have an array of four-byte integers called `iarray`, let us suppose the compiler has chosen to store its first element, `iarray[0]` at byte 640. Then `iarray[1]` will be stored at byte 644, `iarray[2]` will be stored at byte 648, etc:

As always, the compiler decides where to store the array for us.

Byte:	640 641 642 643 644 645 646 647 648 649 650 651 etc.
Used for:	<---- iarray[0] ----> <---- iarray[1] ----> <---- iarray[2] ----> etc.

C treats the name of an array as a shorthand for its address

So in the example above, `iarray` has the value "640". We can even print out the value of `iarray` using the "%p" format used for addresses, as we shall see in the section below.

One result of this is that unlike some more specialised languages, C does not allow whole-array operations:

```
int myarray[N], yourarray[N];
...
yourarray = myarray;          /* Wrong */

for (i = 0; i < N; ++i)      /* Correct */
    yourarray[i] = myarray[i];
```

Stepping over the edge of the cliff

If in our previous example we assume that `iarray` had dimension 10, then the compiler would have reserved forty bytes (640 to 679 inclusive) to store it. This prompts the question: "Exactly what happens if I try to assign a value to the eleventh element, `iarray[10]`?" The answer is that the computer will try to write four bytes to location 680.

This will most likely result in one of two things:

- The variable whose address is 680 will have its value mysteriously changed. This tends to lead to random and confusing behaviour.
- This will be a piece of memory we are not allowed to access and the program will die with a "segmentation fault" or SIGSEGV as it is also called.

Exactly the same will happen if we pass the wrong address to `scanf()`, or we try to access a file we have opened with `fopen()` that we have not checked for being NULL.

What you should remember

- SIGSEGV, and to a lesser extent random behaviour, is most likely due to an array error, a bad address passed to `scanf()`, or using the result of a failed call to `fopen()`.
- The name of an array is the address of the array in memory.

Example 2: reading an array using a function

Having said it would be useful to have a function called `readarray()`, here it is. To demonstrate that it is the address of the array that is passed to a function we print out the value of "numbers" and "input" to demonstrate they are the same.

```
/*
 * Demonstrate passing an array to a function
 */
#include <stdio.h>
#define VALUES 1000000
```

```

void readarray(int input[]);

int main() {
    int numbers[VALUES];

    printf("In main() the array is stored at address %p\n", numbers);
    readarray(numbers);

    printf("The value of numbers[7] is: %d\n", numbers[7]);
    return 0;
}

void readarray(int input[]) {
    int i;
    FILE *infile;

    if ((infile = fopen("mydata.dat", "r")) == NULL ) {
        fprintf(stderr, "Failed to open mydata.dat for input\n");
        return;
    }

    printf("In readarray() the array is stored at address %p\n", input);
    for (i = 0; i < VALUES; ++i ) {
        fscanf(infile, "%d\n", &input[i]);
    }
}

```

The output is:

```

In main() the array is stored at address 0x7fff0f495e40
In readarray() the array is stored at address 0x7fff0f495e40
The value of numbers[7] is: 7

```

(Note that addresses are printed in hexadecimal.) Of course, the exact address printed out may be different each time. When I run it again a few seconds later I get:

```

In main() the array is stored at address 0x7fffe9c3d9f0
In readarray() the array is stored at address 0x7fffe9c3d9f0
The value of numbers[7] is: 7

```

Multi-dimensional arrays

These work in pretty much the same way. The following example declares and initialises a two dimensional array.

```

#include <stdio.h>

#define NX 3
#define NY 2

int main() {
    int grid[NX][NY] = { { 1, 2 }, {3, 4 }, { 5, 6 } };
    int x, y;

    for (x = 0; x < NX; ++x)
        for (y = 0; y < NY; ++y)
            printf("grid[%d][%d] is %d\n", x, y, grid[x][y]);

    return 0;
}

```

The output is:

```

grid[0][0] is 1
grid[0][1] is 2

```

In some situations it may be more natural to use `NROWS`, `NCOLS`, `row` and `col` instead of `NX`, `NY`, `x` and `y`.

Or if we had a two-dimensional array of planetary data for different stellar systems we might use `star` and `planet`.

```

grid[1][0] is 3
grid[1][1] is 4

grid[2][0] is 5
grid[2][1] is 6

```

Run-time array sizes

The original version of the C language required the dimensions of an array to be constants, defined when the program was written but this restriction has now been relaxed. It is occasionally useful to be able to write:

```

void myfun(int n) {
    double x[n];
    ...
}

```

Passing multidimensional arrays to functions

When passing multi-dimensional array to functions we have to tell the function the dimensions of the array (apart from the leading dimension). This isn't a problem if we are using symbolic constants, but if they are dimensioned at run-time we have to pass the dimensions as additional arguments, **before** the array name in the argument list:

The following is a very simple matrix-multiply function.

Later we shall see an even more flexible way of doing this using dynamic memory allocation.

```

/*
 * Multiply two matrices together: A = B x C
 */
void matmul(int nx, int ny, int nz, double a[nx][ny], double b[nx][nz],
            double c[nz][ny]) {
    int x, y, z;

    for (x = 0; x < nx; ++x) {
        for (y = 0; y < ny; ++y) {
            a[x][y] = 0.0;
            for (z = 0; z < nz; ++z)
                a[x][y] += b[x][z] * c[z][y];
        }
    }
}

```

As always when using arrays, if we were to make a mistake and end up with the function thinking that the dimensions of the array were different from those it was originally declared with, we would be in serious trouble.

Summary

- Use arrays for vectors and matrices and to avoid "copy-and-paste" loops.
- **The elements of an array of size N are array[0] to array[N-1] inclusive.**
- Use symbolic constants for array sizes, not "magic numbers" such as "10". (Array sizes can be integer expression, evaluated at run-time.)
- Just like variables, an array can be initialised when we declare it.
- Randomly-changing variables or SIGSEGV are likely to be due to going over the end of an array, bad arguments to `scanf()` or failing to check for failed calls to `fopen()`.
- The `for(Initialise; Test; Increment)` loop can be used to loop over the elements of an array.
- "When an array name is passed to a function, what is passed is the location of the beginning of the array." [K&R]. So changing an array element inside a function changes the original.
- If we do pass a multi-dimensional array to another function we also have to tell that function the dimensions of the array, other than the left-most one.

After the lecture

- Revise the key points.
- This week the larger examples are in the main body of the lecture. Look through them identifying how the key points apply.

