Arrays

Sums

Here's a (poor) program to add three numbers

add.c

```c
/* Add three numbers */
#include <stdio.h>

int main() {
  int numbers[3];
  numbers[0] = 16;
  numbers[1] = 12;
  numbers[2] = 14;
  int sum = 0;
  for (int i=0; i<3; i++) sum = sum + numbers[i];
  printf("sum = %d\n", sum);
}
```

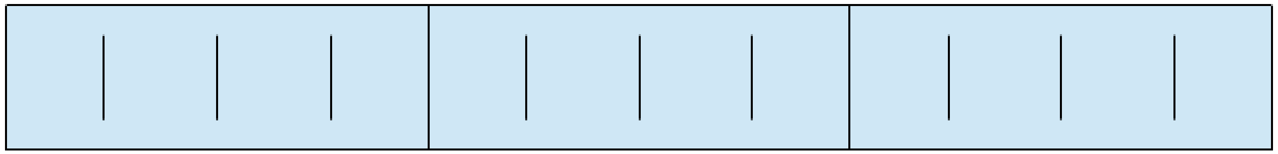
Declaring arrays

The array declaration is:

```
int numbers[3];
```

The type (`int`) is the type of each element

The length (`3`) is the number of elements



Amount of memory is `sizeof(numbers)`
= `3 * sizeof(int)` = `3*4` = `12`

Variable length arrays

The length of an array can be a variable (in C11)

```
int calculate(int n) {
  int numbers[n];
  ...
}
```

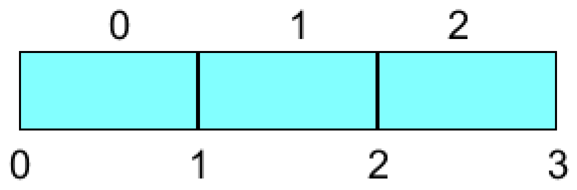
The variable can even be read in from the user or a file

The length of an array can't change after it is declared (it only varies between function-calls or program-runs)

Starting from zero

Sometimes, programmers focus on the elements of an array, sometimes on the positions, i.e. gaps in between

That is like counting or like measuring:



Counting and measuring both start at `0` so that there are fewer problems (out-by-one errors or special cases)

Example: substring

Suppose you want to write a substring function

How do you say where the substring starts and ends?

If you count the characters, the usual convention is "from the `i`'th to the `j`'th characters inclusive"

But the length of the substring is then `j-i+1`, and empty substrings must have `j` less than `i`

With `0`-based positions, the length is `j-i` and empty substrings have `j==i`

Bugs

There are fewer problems when indexing from `0`, but it is still easy to make mistakes, especially ones like:

```c
int numbers[3];
numbers[3] = 42;
```

The element `numbers[3]` doesn't exist, so either this updates some other variable, or the memory doesn't even belong to your program

C does not automatically detect this, so the program may run on and produce stupid results, or crash

Segmentation faults

A segmentation fault or ***segfault*** is when your program tries to access memory which doesn't belong to it

segfault.c

```
/* Generate a segfault. */
#include <stdio.h>

int main() {
    int numbers[3] = {3, 4, 5};
    int n = 1000000000;
    printf("%d\n", numbers[n]);
}
```

This program will ***probably*** end in a segfault

Undefined behaviour

There are many situations besides running off the end of an array which result in *undefined behaviour* (i.e. crashes or random results)

The C standard does ***not*** insist that a compiler or run-time system detects these errors, because that would make it a lot less efficient

It used to be a big problem for C programmers

But now, you can use the `-fsanitize=undefined` option (which works on Linux/MacOs and half works on Windows)

Debugging

The C language ***cannot*** warn you about all your bugs

Often, your program does the wrong thing, and ***you*** have to find the bug

This skill, see aside: debugging, is an essential one in programming

It is almost impossible to teach in lectures - it is lab experience that counts - ask if you get stuck

It is really important to reduce your debugging to, say, less than 50% of your development time - by practice, practice, practice

Initialising arrays

For efficiency, arrays are not initialised, and their contents are rubbish (whatever happens to be in the allocated memory)

So, each entry must be written before it is read

Constant-length arrays can be initialised compactly:

```
int numbers[3] = {16, 12, 14};
```

Note: when curly brackets are used for initialisation, you need a semicolon after the close bracket, unlike with block brackets

The compiler can count

You can write this:

```
int numbers[] = {16, 12, 14};
```

The compiler can work out the length of the array (3)

Passing arrays

Let's try a separate function for summing:

sum.c

```
// Demo: pass an array
#include <stdio.h>

// Add three numbers
int sum(int items[3]) {
   int sum = 0;
   for (int i=0; i<3; i++) sum = sum + items[i];
   return sum;
}

int main() {
   int numbers[3] = {16, 12, 14};
   printf("sum = %d\n", sum(numbers));
}
```

Array lengths

The `sum` function would be much better if it could accept an array of any length

But there is no way to find the length of an array

Instead, you have to remember how big it was when you created it, and store that length somewhere else

Generalized sum

That means the length of an array has to be passed to a function, as well as the array itself:

```
// Demo: pass an array                                sum2.c
#include <stdio.h>

// Add up an array of numbers
int sum(int n, int items[n]) {
  int sum = 0;
  for (int i=0; i<n; i++) sum = sum + items[i];
  return sum;
}

int main() ...
```

Passing array lengths

The argument notation `(int n, int items[n])` is a recent one, which only works if `n` is ***before*** `items[n]` in the argument list

Also common is `(int n, int items[])` because the compiler doesn't need to know the length of `items`, it just needs to do what it is told with `n`

Also common is `(int n, int *items)` (see later chapter) because the compiler treats "array" and "pointer to start of array" as the same thing (almost)

Pass by value or reference?

Does this work?

```
/* Test array-passing */                                   pass.c
#include <stdio.h>

// Initialise elements of array
void init(int n, int items[n], int v) {
  for (int i=0; i<n; i++) items[i] = v;
}

int main() {
    int numbers[3] = { 0, 0, 0 };
    init(3, numbers, 42);
    printf("n0 = %d\n", numbers[0]);
}
```

Pass by value

Ordinary variables are passed by value

That means, the variable (or constant or calculated expression) is ***copied*** into the argument variable

Any changes made to the argument variable have no effect on the original

```
int square(int x) { x = x*x; return x; }
...
int n = 7, s = square(n);
printf("%d %d\n", n, s);
```

The function `square` has no effect on `n`

Pass by reference

Arrays are passed by reference

That means the argument variable is made to refer to the original array, because copying a large array would be expensive

That means that if a function changes any of the elements of the array, those changes are seen in the caller's array (the array has two names)

Returning arrays

Arrays can't be returned from functions

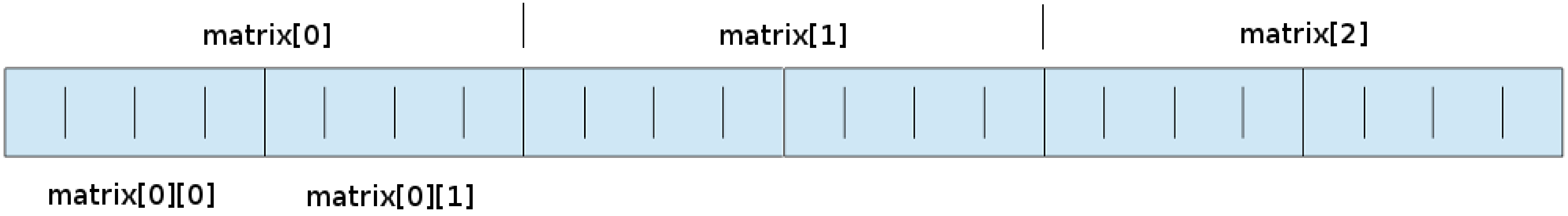
Pointers to arrays can be returned (see later chapter)

For now, to design a function which adds two vectors, you can pass a third argument to hold the result, which works because of pass-by-reference

```
void add(int n, int a[n], int b[n], int c[n]) {
    ...
}
```

Multi-dimensional arrays

A two dimensional array in C is just an array of arrays:



```
int matrix[3][2] = {{1,2},{3,4},{5,6}};
printf("bottom right = %d\n", matrix[2][1]);
```

Printing a matrix

When passing a multi-dimensional array, the compiler *does* need to know the length of each dimension (except the first), so the modern notation is essential:

```
void print(int h, int w, int matrix[h][w]) {
    for (int r=0; r<h; r++) {
        for (int c=0; c<w; c++) {
            if (c > 0) printf(" ");
            printf("%d", matrix[r][c]);
        }
        printf("\n");
    }
}
```

Concerns

Most programmers worry about allocating large arrays, because they are allocated on the stack, and the stack may have a limited size (see memory chapter)

Also, the strict lifetime of an array (allocated on a call and deallocated on a return) is not flexible enough

So, sooner or later, you have to switch to dynamic allocation (see later chapter) - most programmers use dynamic allocation most of the time