Welcome!

COMP1511 18s1

Programming Fundamentals

COMP1511 18s1

— Lecture 9 —

More Strings

~~Andrew Bennett~~

~~<andrew.bennett@unsw.edu.au>~~

Jashank Jeremy

<jashank.jeremy@unsw.edu.au>

Before we begin…

introduce yourself to the person sitting next to you

why did they decide to study **computing**?

Overview

after this lecture, you should be able to…

understand how to **initialise** an **array**

understand the various ways to **initialise** a **string**

have a basic understanding of functions from **string.h**

understand the basics of working with **command line arguments**
(i.e. **argc** and **argv**)

(**note**: you shouldn't be able to do all of these immediately after watching this lecture. however, this lecture should (hopefully!) give you the foundations you need to develop these skills. remember: programming is like learning any other language, it takes consistent and regular practice.)

Admin

Don’t panic!

week 4 weekly test due friday

don’t be scared!

friday this week is a **public holiday**

if you have a friday tutelab: see course website for details

this week’s **lab** due **friday midsem break**

Friday 6th April

assignment 1 due **sunday midsem break**

you **need** to start **now**, if you haven’t already

don’t forget about **help sessions**!

see course website for details

Initialising Arrays

initialising arrays is **important**

(remember yesterday?)

Revisiting: Uninitialised Arrays

the array has not been **initialised**

```
int array[SIZE];

int i = 0;
while (i < SIZE) {
    printf("%d\n", array[i]);
    i++;
}
```

what should **printf** print?

this is **undefined behaviour**

(there's no rule in C about what should happen

Revisiting: Uninitialised Arrays

solution: initialise the array first

(note: you could also initialise all the values in a loop)

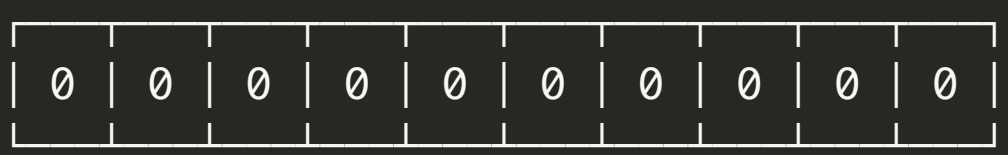
```c
int array[SIZE] = {0};

int i = 0;
while (i < SIZE) {
    printf("%d\n", array[i]);
    i++;
}
```

Initialising Arrays

there are several ways to initialise an array

using an **array initialiser**

```
// array will be filled with all zeroes
int array[SIZE] = {0};
```



Initialising Arrays

there are several ways to initialise an array

using an **array initialiser**

```
// array will be initialised with 1, 2, 3, 4, 5, then the rest 0
int array[SIZE] = {1, 2, 3, 4, 5};
```

Initialising Arrays

there are several ways to initialise an array

using a **loop**

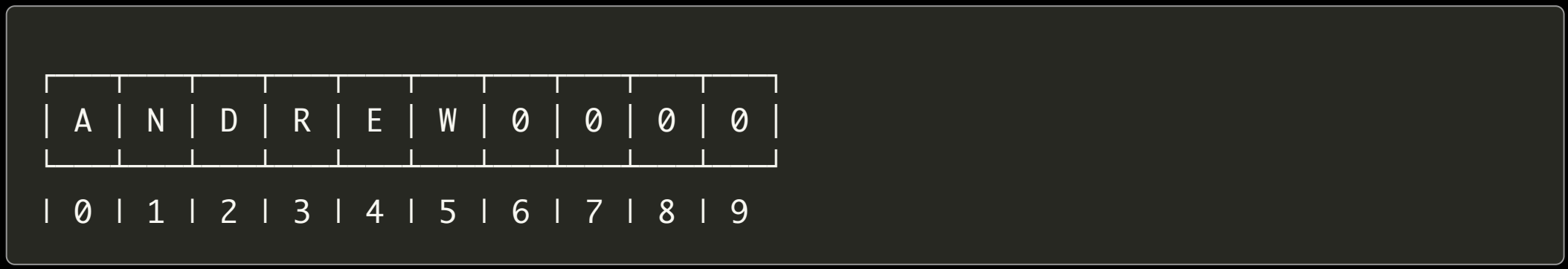
```
int i = 0;
while (i < SIZE) {
    array[i] = i;
}
```

(this is more **flexible** and allows you to initialise with values of your choice)

Initialising Strings

we can initialise strings in a similar way

```
char name[SIZE] = {'A', 'N', 'D', 'R', 'E', 'W'};
```

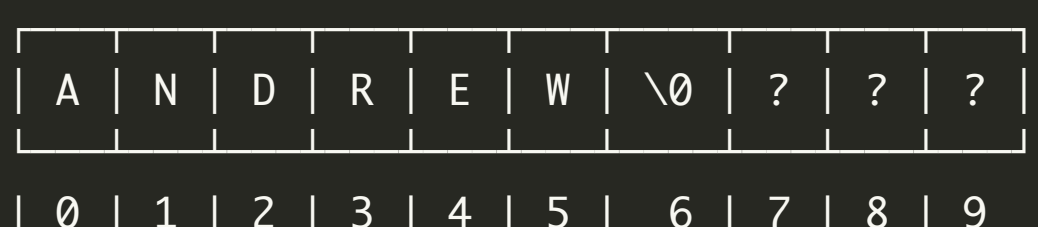


(remember: the remaining elements are initialised with zeroes)

Initialising Strings

there's a short-hand in C

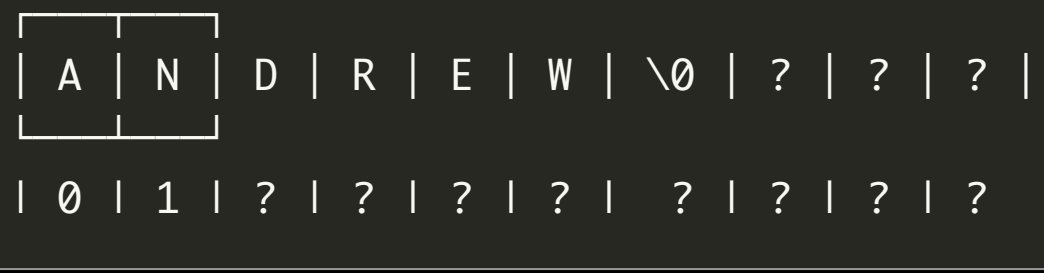
```
char name[SIZE] = "ANDREW";
```



Initialising Strings

what happens if we try to set more than will fit?

```
#define SIZE 2
char name[SIZE] = "ANDREW";
```



Initialising Strings

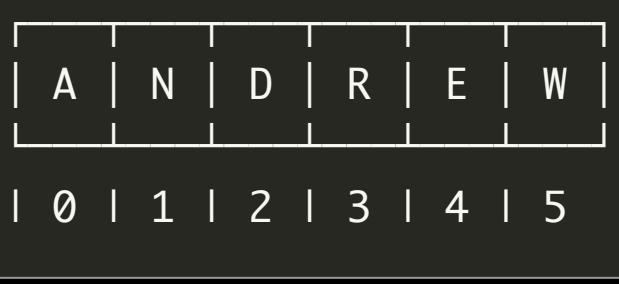
if we leave the size out, C will automatically make it big enough

```
char name[] = "ANDREW";
```

Initialising Strings

if we leave the size out, C will automatically make it big enough

```
char name[] = {'A', 'N', 'D', 'R', 'E', 'W'};
```



what's the problem here?

try it and see!

Initialising Strings

if we leave the size out, C will automatically make it big enough

```
char name[] = {'A', 'N', 'D', 'R', 'E', 'W', '\0'};
```

introducing:
command line arguments

Command Line Arguments

"0 or more" strings specified when the program runs

you've already seen these

```
dcc -o hello hello.c
```

here, **dcc** is being run with 3 command line arguments:
`-o`, `hello`, `hello.c`

Command Line Arguments

“0 or more” strings specified when the program runs

```
./hello
```

the program **hello** has 0 command line arguments

Command Line Arguments

“0 or more” strings specified when the program runs

```
./hello some thing goes here
```

the program **hello** has 4 command line arguments

Command Line Arguments

we can access these in our program by changing the **signature** of the **main** function

```
int main (int argc, char *argv[]) {
    // code goes here
}
```

argc stores the **number** of arguments

argv stores the **contents** of the arguments

Command Line Arguments

```
./program hello there
```

this has two arguments: “hello” and “there”

```
int main (int argc, char *argv[]) {
    // argc is 2
    printf("%d\n", argc);

    // print out all of the arguments
    int i = 0;
    while (i < argc) {
        printf("Argument %d is: %s\n", i, argv[i]);
        i++;
    }
}
```

argc stores the **number** of arguments

argv stores the **contents** of the arguments

fgets

`fgets(array, array size, stream)` reads a line of text

array - char array in which to store the line

array size - the size of the array

stream - where to read the line from, e.g. stdin

fgets won't try to store more than **array size** chars in the array

never use the function `gets`! (why?)

string.h

```
#include <string.h>
// string length (not including '\0')
int strlen(char *s);
// string copy
char *strcpy(char *dest, char *src);
char *strncpy(char *dest, char *src, int n);
// string concatenation/append
char *strcat(char *dest, char *src);
char *strncat(char *dest, char *src, int n);
```

string.h

```c
#include <string.h>
// string compare
int strcmp(char *s1, char *s2);
int strncmp(char *s1, char *s2, int n);
int strcasecmp(char *s1, char *s2);
int strncasecmp(char *s1, char *s2, int n);
// character search
char *strchr(char *s, int c);
char *strrchr(char *s, int c);
```