ENGG1003 - Friday Week 5

Arrays and Functions: Together at Last!

Does anyone even read the title page?

Also: Maybe Strings

Brenton Schulz

University of Newcastle

March 28, 2019

The Story So Far

- Course summary:
  - Flow control
    - `if()`
    - `while()`
    - `for()`
    - `switch()`
  - Variables and data types
  - Functions
  - Arrays
- Today: Arrays and functions together
  - Subtext: Pointers
- Today (maybe): Strings
- Tuesday: File input-output (I/O)

Programming Assignment And Quiz

- The programming assignment will use everything from the previous slide
- The quiz can include everything up to, and including, the Week 5 Tuesday lecture
  - Held in Friday 9-10am lecture
    - Exact duration TBA
  - It will be hand written
    - Yes, *real paper*
  - Mix of:
    - Multiple choice
    - Code reading & analysis
    - Short code writing (1-3 lines)
  - You will **not** be asked to write out a whole program by hand

Arrays and Functions

- On Tuesday:
  - Studied arrays
  - Studied functions
  - Didn't mix them

Arrays and Functions

- On Tuesday:
  - Studied arrays
  - Studied functions
  - Didn't mix them
- There are two ways to pass arrays to functions:

Arrays and Functions

- On Tuesday:
  - Studied arrays
  - Studied functions
  - Didn't mix them
- There are two ways to pass arrays to functions:
  - Pass an array element, eg:

```c
int function(int x);
// ...
int array[12];
// ...
function(array[6]);
```

Arrays and Functions

- On Tuesday:
  - Studied arrays
  - Studied functions
  - Didn't mix them
- There are two ways to pass arrays to functions:
  - Pass an array element, eg:

```c
int function(int x);
// ...
int array[12];
// ...
function(array[6]);
```

  - Give a function a *pointer* to an array
    - Ok, lets break this one down a bit...

Arrays and Functions

- Firstly: why don't we pass a whole array?

Arrays and Functions

- Firstly: why don't we pass a whole array?
  - Arrays can be *huge*
  - Passing a whole array *copies* everything
  - This is a bad idea so C doesn't support it
  - (Advanced) Arguments are put to the *stack*
    - Google stack Vs heap memory allocation for more information. This is beyond ENGG1003.

Arrays and Functions

- Firstly: why don't we pass a whole array?
  - Arrays can be *huge*
  - Passing a whole array *copies* everything
  - This is a bad idea so C doesn't support it
  - (Advanced) Arguments are put to the *stack*
    - Google stack Vs heap memory allocation for more information. This is beyond ENGG1003.
- Instead, C passes a *pointer*
  - This is the *memory address* of the array's start

Arrays and Functions

- Firstly: why don't we pass a whole array?
  - Arrays can be *huge*
  - Passing a whole array *copies* everything
  - This is a bad idea so C doesn't support it
  - (Advanced) Arguments are put to the *stack*
    - Google stack Vs heap memory allocation for more information. This is beyond ENGG1003.
- Instead, C passes a *pointer*
  - This is the *memory address* of the array's start
  - In C, the array's name is equivalent to `&name[0]`

Arrays in Memory

- Review: When we declare an array, eg,

```c
int x[20];
```

  the compiler allocates 20*sizeof(int) = 80 bytes to store it

- The *memory address* of `x[0]` is some seemingly random number, `p`
- Other elements are stored in sequential memory addresses:
  - The address of `x[1]` is `p + 4`
  - The address of `x[i]` is `p + i*4`

Arrays in Memory

- Therefore, to access any element of an array all we need is:
  - A pointer to the first element
  - Knowledge of the arrays *data type*
    - Specifically: the type's *size*

Arrays in Memory

- Therefore, to access any element of an array all we need is:
  - A pointer to the first element
  - Knowledge of the arrays *data type*
    - Specifically: the type's *size*
- So that's what we do with functions:
  - The function argument is a *pointer* to a *data type*

Arrays in Memory

- Therefore, to access any element of an array all we need is:
  - A pointer to the first element
  - Knowledge of the arrays *data type*
    - Specifically: the type's *size*
- So that's what we do with functions:
  - The function argument is a *pointer* to a *data type*
- C syntax:

```
return_type function_name(data_type *varName);
```

- Key syntax element: the ⋆ character

Arrays in Memory

- Therefore, to access any element of an array all we need is:
  - A pointer to the first element
  - Knowledge of the arrays *data type*
    - Specifically: the type's *size*
- So that's what we do with functions:
  - The function argument is a *pointer* to a *data type*
- C syntax:

```
return_type function_name(data_type *varName);
```

- Key syntax element: the $\star$ character
- Inside the function use `var[i]` syntax

Key Points

- Because arrays are passed via a pointer the function gets *the actual array*
- Modifying the array in the function modifies the original variable
- You don't *need* a return value
    - In a technically incorrect way: all the array's elements are "returned"

Example

- Write a function which zeros the first N elements of an array of `ints`
  - Function prototype:

Example

- Write a function which zeros the first N elements of an array of `ints`
  - Function prototype:
    - `void zero(int *x, int N);`

Example

- Write a function which zeros the first N elements of an array of `int`s
  - Function prototype:
    - `void zero(int *x, int N);`
  - The value of `N` is needed because C won't tell you how long an array is *within the context of the function*
    - (Advanced) `sizeof(x)` will just be the size of the pointer - 4, or 8 bytes

Example

- Function definition:

```c
// Zeros first N elements of x
void zero(int *x, int N) {
  int i; // Array index loop counter
  for(i = 0; i < N; i++)
    x[i] = 0; // Use array syntax
  return; // Optional
}
```

Other Examples

- Lets write and test these live...
- Write a function which:
    - Returns the sum of an array of length N
    - Returns the maximum value in an array of length N
    - Fills an array with integers between two given numbers `min` and `max`
        - Prototype:

```
void countArray(int *x,
                int min, int max);
```

Strings

- A *string* is the “data type” which stores human-readable text
- C does not have a `string` data type
  - Most newer languages do, though
- In C strings are stored in arrays of type `char`
  - Their “length” is defined by a terminating zero

String Syntax

- C strings are arrays of type `char`
- They are declared with normal array syntax:

```c
char name[200];
```

- The “size” of a string is known as the *length*
- Strings get terminated with a 0
  - Ok, technically NULL but its just a zero in memory
  - Often NULL is written \0
- The NULL termination defines the length

Strings in Memory

- Each character is a single byte
- The terminating NULL is also a single byte
  - Be aware of this when declaring array sizes
- Everything beyond the NULL is “garbage”
  - Doesn’t matter what the array size is
- The string "hello” would be stored as:

(Addresses are made up numbers)

Address:	10	11	12	13	14	15	16	17
Data:	??	h	e	l	l	o	\0	??