ENGG1003 - Friday Week 5

Arrays and Functions: Together at Last!

Does anyone even read the title page?

Also: Maybe Strings & ASCII Codes

Brenton Schulz

University of Newcastle

March 29, 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
    - 40 mins: 9:10am - 9:50m
  - 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, `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`
- `p` is a *byte* address
- 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 a given element, `i`, of an array all we need is:
  - A pointer, `p` to the first element
  - Knowledge of the arrays *data type*
    - Specifically, the type's *size*
  - The calculation result of `p + i*size`

Arrays in Memory

- Therefore, to access a given element, `i`, of an array all we need is:
  - A pointer, `p` to the first element
  - Knowledge of the arrays *data type*
    - Specifically, the type's *size*
  - The calculation result of `p + i*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 a given element, `i`, of an array all we need is:
  - A pointer, `p` to the first element
  - Knowledge of the arrays *data type*
    - Specifically, the type's *size*
  - The calculation result of `p + i*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 a given element, `i`, of an array all we need is:
  - A pointer, `p` to the first element
  - Knowledge of the arrays *data type*
    - Specifically, the type's *size*
  - The calculation result of `p + i*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
- 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 `int`s
  - 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);
      ```
    - eg: countArray(x, 10, 15) sets:
      `x[] = {10, 11, 12, 13, 14, 15}`

Strings

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

String Syntax

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

```
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 length is the number of bytes from (and including) the "start" pointer and the \0

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	??

Using Strings

- String initialisation uses the syntax:

```
char str[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

Using Strings

- String initialisation uses the syntax:

```
char str[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

- Terrible, isn't it?

Using Strings

- String initialisation uses the syntax:

```c
char str[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

- Terrible, isn't it?
- If the string is *constant* you can do this:

```c
char str[] = "This is a constant string.";
```

  - Attempting to modify `str[]` will cause a crash
  - The compiler automatically inserts the `\0`
  - (Advanced) Strings between " and " are stored in the "program memory" and can't be modified for security reasons

Constants

- Aside: any variable which must not be modified can be declared `const`:

```
const char str[] = "This is a help message.";
```

- The `const` keyword causes a compiler error, instead of a segmentation fault, if you try to modify the variable
- You can do this to any data type, eg:

```
const float pi = 3.14159;
```

String Usage

- Normally strings are not initialised with `{'a', 'b',}` syntax
- Command line programs use a lot of constant strings
  - Text inside `printf()` is a constant string
- Most strings are read from the user or a file
  - In embedded systems they also come from communications peripherals like a UART

String Format Specifiers

- To `printf()` or `scanf()` a string use the `%s` format specifier
- Eg:

```c
#include <stdio.h>
main() {
  char str[] = "Hello world!";

  // NB: Passing array pointer to function
  // just uses the array name as argument
  printf("%s\n", str);
}
```

Strings and `scanf()`

- Because an array name is a pointer to the first element **do not use &**

```c
char str[1024];
scanf("%s", str); // NO & SYMBOL
```

Strings and `scanf()`

- Because an array name is a pointer to the first element **do not use &**

```c
char str[1024];
scanf("%s", str); // NO & SYMBOL
```

- How much data does `%s` read?

Strings and `scanf()`

- Because an array name is a pointer to the first element **do not use &**

```c
char str[1024];
scanf("%s", str); // NO & SYMBOL
```

- How much data does `%s` read?
- Lets test with an example

```c
#include<stdio.h>
int main() {
    char str[1024];
    scanf("%s", str);
    printf("Read: %s\n", str);
}
```

Strings and `scanf()`

- Experiment results:
  - `%s` stops at the first whitespace character
  - It ignores whitespace
  - Interpretation: `%s` reads a single word or number
- This changes if more complicated "pattern matching" is included in the `scanf()` argument
  - Beyond ENGG1003

ASCII Codes

- In C, constant letters in code are typed: `'a'`
- The single quote indicates that it is a *literal* letter, not a string

ASCII Codes

- In C, constant letters in code are typed: `'a'`
- The single quote indicates that it is a *literal* letter, not a string
- But what is *actually* stored? Doesn't `char` just store a number from -128 to +127?

ASCII Codes

- In C, constant letters in code are typed: `'a'`
- The single quote indicates that it is a *literal* letter, not a string
- But what is *actually* stored? Doesn't `char` just store a number from -128 to +127?
- Yes! The ASCII standard converts "letters" to numbers

ASCII Codes

- The ASCII standard allocates a number to all letters, numbers, punctuation characters, and several “control” characters
- ASCII is used almost everywhere
  - The unicode standard UTF-8 is a superset of ASCII
- Lets check one out here
- Knowledge of ASCII and `char` processing in C is necessary for Programming Assignment 1

Char Variables

- There are two ways to interpret a `char` variable:
  - As a text character
  - As a number
- The `%c` format specifier tells `printf()` and `scanf()` to convert between ASCII characters and numbers
- Eg, this will read a character from `stdin` and store its ASCII value in `c`:

```
char c;
scanf("%c", &c)
```

Char Variables

- What happens if you enter the number 5?

```
char c;
scanf("%c", &c)
```

Char Variables

- What happens if you enter the number `5`?

```c
char c;
scanf("%c", &c)
```

- It will store the number 53, as that is the ASCII code for '5'

ASCII Letters

- The project requires you to process text and identify letters
- The following table shows the numerical values which letters can occupy under the ASCII standard:

A	65	a	97
B	66	b	98
C	67	c	100
...		...	
Y	89	y	121
Z	90	z	122

Char Variables

- The numerical value of a character can be printed with `%d` and a cast:

```c
char c;
printf("%d", (int)c);
```

- Characters can be used in *arithmetic* without a problem, eg:

```c
char c;
scanf("%c", c);
c = c - 65;
```

Char Variables

- The numerical value of a character can be printed with `%d` and a cast:

```c
char c;
printf("%d", (int)c);
```

- Characters can be used in *arithmetic* without a problem, eg:

```c
char c;
scanf("%c", c);
c = c - 65;
```

Char Variables

- The code "`c = c - 65`" will convert each letter of the alphabet to a number with the allocation:

$$
\begin{aligned}
A &= 0 \\
B &= 1 \\
C &= 2 \\
&\dots \\
Z &= 25.
\end{aligned}
$$

Char Variables

- The code "`c = c - 65`" will convert each letter of the alphabet to a number with the allocation:

$$
\begin{aligned}
A &= 0 \\
B &= 1 \\
C &= 2 \\
&\dots \\
Z &= 25.
\end{aligned}
$$

- You use this in the project

`char` Example

Write a C program which reads a single char from the user and uses it to select from a text-based menu.

This is useful for Programming Assignment 1

`char` Example

Write a C program which reads a single char from the user and uses it to select from a text-based menu.

This is useful for Programming Assignment 1

- We should `printf()` a menu
- A single character should be read back
- Use `switch` to call off the appropriate function

`char` Example

- We can print a menu like this:

```
printf("Please select an option: \n");
printf("a) Start a new game\n");
printf("b) Load a saved game\n");
printf("c) Options\n");
printf("d) Quit\n\n");
printf("Selection: ");
```

char Example

- We can print a menu like this:

```c
printf("Please select an option: \n");
printf("a) Start a new game\n");
printf("b) Load a saved game\n");
printf("c) Options\n");
printf("d) Quit\n\n");
printf("Selection: ");
```

- And read the user's input with:

```c
char c;
scanf("%c", &c); // &c, not a string
```

`char` Example

- We need to read user input *at least once*
- The user could make a mistake
- Lets use a `do { } while()`

`char` Example

- We need to read user input *at least once*
- The user could make a mistake
- Lets use a `do { } while()`
- What would be used in the condition?

`char` Example

- We need to read user input *at least once*
- The user could make a mistake
- Lets use a `do { } while()`
- What would be used in the condition?
- `c` needs to be `a`, `b`, `c`, or `d` to continue

`char` Example

- We need to read user input *at least once*
- The user could make a mistake
- Lets use a `do { } while()`
- What would be used in the condition?
- `c` needs to be `a`, `b`, `c`, or `d` to continue
- Couple of options:
  - Naive solution:

```
while(c != 'a' && c != 'b' && c != 'c' &&
      c != 'd');
```

  - Use knowledge of ASCII codes:

```
while(c < 'a' || c > 'd');
```

char Example

- Once input is taken lets use switch():

```
switch(c) {
  case 'a': newGame(); break;
  case 'b': loadGame(); break;
  case 'c': options(); break;
  case 'd': quit(); break;
  default: printf("Unknown option %c\nPlease
    enter a, b, c, or d\n");
}
```

`char` Example

Lets see the full program in Che...