COMP1511/1911 Programming Fundamentals

Week 5 Lecture 1

Lecture Program 2D Arrays of structs
Multi-file Programs

Last Week

- 2D Arrays
- Strings
- We did not get up to arrays of strings or command line arguments

Public Holiday

Tuts/labs

- Students in monday classes please book and attend another class for week 5. Link to book here Access code is COMP1511 timetable
- Lab week 4 deadline: Week 5 Tuesday 8pm

Today’s Lecture

- Revision: A bigger 2D array of structs with enums program!
  - mud_and_bones.c
  - Putting together concepts needed in assn1
  - Style tips for assn 1
- Recap strings
- Array of strings
- Command line args
- Multi-file Projects

Link to Week 5 Live Lecture Code

https://cgi.cse.unsw.edu.au/~cs1511/24T3/live/week_5/

Problem Time
Put together the important concepts needed for assn1

Problem Time: Mud and Bones

We have the following “game”.

- A dog (the player) is moving around on a map
- The locations on the map contain either grass, mud or water.
- They may also contain a bone.
- The dog can move around the map to collect bones
- However if he was in a location with mud on the previous turn he will spread the mud to the grass if he lands on grass
- If he was in a location with water on the previous turn he will wash the mud off the grass if he lands on mud

Problem Time: Mud and Bones

There is no winning in this “game”
The player presses Ctrl^D to end the game!

Warning: This is not how mud, water and grass works in real life…
don’t try this with your own dog.

Problem Time: Mud and Bones

Important types and constants given to you for this code

```
#define MAP_ROWS 8
#define MAP_COLUMNS 8
enum ground_type {
    GRASS,
    WATER,
    MUD
};
```

```
enum item_type {
    EMPTY,
    BONE
};
struct location {
    enum item_type item;
    enum ground_type ground;
};
```

The Map: 8x8 2D array of struct location

```
struct location map[MAP_ROWS][MAP_COLUMNS];
```

	Col 0	Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7
Row 0								
Row 1								
Row 2								
Row 3								
Row 4								
Row 5								
Row 6								
Row 7								

The Map: 8x8 2D array of struct location

```c
struct location map[MAP_ROWS][MAP_COLUMNS];
```

	Col 0	Col 1	Col 2
Row 0	EMPTY GRASS	BONE MUD	EMPTY GRASS
Row 1	EMPTY WATER	BONE GRASS	EMPTY GRASS
Row 2	EMPTY GRASS	EMPTY GRASS	EMPTY GRASS
Row 3	EMPTY GRASS	EMPTY GRASS	EMPTY GRASS

If we zoom into a section of the map, we can see each one is a struct location with an item type and a ground type

The Map: 8x8 2D array of struct location

```
struct location map[MAP_ROWS][MAP_COLUMNS];
```

	Col 0	Col 1	Col 2
Row 0	EMPTY GRASS	BONE MUD	EMPTY GRASS
Row 1	EMPTY WATER	BONE GRASS	EMPTY GRASS
Row 2	EMPTY GRASS	EMPTY GRASS	EMPTY GRASS

In this example

map[0][1].item
has the value BONE

map[0][1].ground has the value MUD

Problem Time: Mud and Bones

Provided Function Prototypes

```c
void initialise_map(struct location map[MAP_ROWS][MAP_COLUMNS]);
void print_map(
    struct location map[MAP_ROWS][MAP_COLUMNS],
    int dog_row,
    int dog_col,
    int num_bones,
    int mud_spread
);
```

Mud and Bones Stage 1

- Create a map variable
- Call initialise on the map
- Call print_board, passing in ILLEGAL_INDEX for dog_row and dog_col and 0 for bone_count and mud_count
- Initialise data:
    - scan in co-ordinates from the user and set the dog’s starting position. If illegal, set to (0, 0)
    - initialise bone_count and mud_count to 0
    - Print the board!

Mud and Bones Stage 2

In a loop that ends with Ctrl-D (there is no winning)

- Allow the user to enter ‘w’ ‘a’ ‘s’ ‘d’ to move the dog around the map (other inputs are ignored). We will not implement checking bounds of array. So our program may crash :(
- Update the changes in ground_type based on the dog’s movement through water and mud
- Increment the bone count and remove bones from the map once found. Print out “Yum!”
- Print the map after each valid move

Assignment 1 Style Tips

Follow the style guide, but some simple things to watch out for:

- Functions
- #defines constants for magic numbers including ‘w’ etc
- Comments
- line length

Get feedback from

- style checker
- checking the style guide
- asking your tutor or a help session tutor to give feedback

Feedback Please!

Week 5 Part 1

Your feedback is valuable!

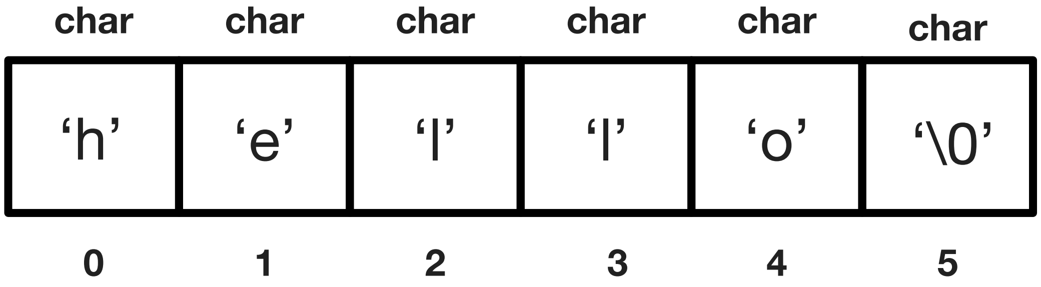
If you have any feedback from today's lecture, please follow the link below or use the QR Code.

Please remember to keep your feedback constructive, so I can action it and improve your learning experience.

https://forms.office.com/r/BKnXfy15i7

Strings recap: What are they?

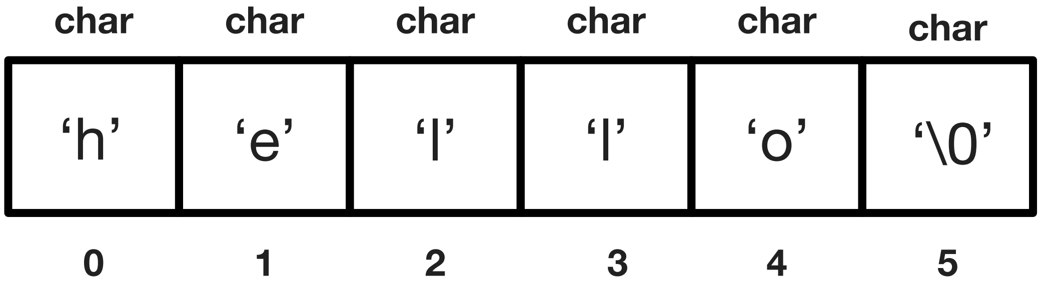
- Strings are a collection of characters
- In C a string is
  - an array of `char`
  - that ends with a special character `'\0'` (null terminator)



char	char	char	char	char	char
‘h’	‘e’	‘l’	‘l’	‘o’	‘\0’
0	1	2	3	4	5

Strings: How do we initialise them?

```
// the painful way
char word[] = {'h','e','l','l','o','\0'};
// the more convenient way which does the same thing
char word[] = "hello";
```

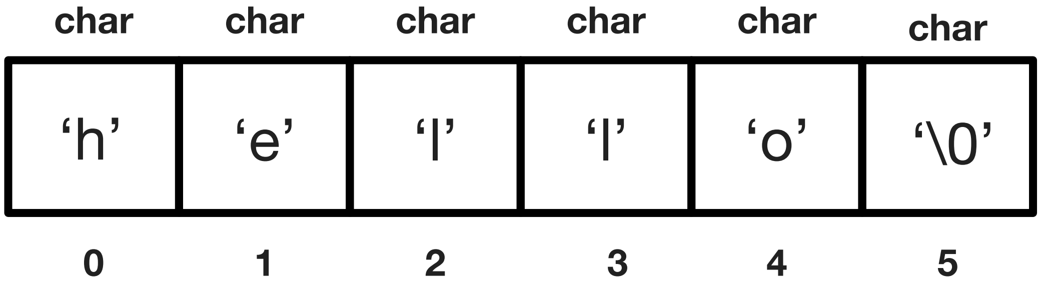


char	char	char	char	char	char
‘h’	‘e’	‘l’	‘l’	‘o’	‘\0’
0	1	2	3	4	5

Printing Strings

```
char word[] = "hello";
int i = 0;
while (word[i] != '\0') {
    printf("%c", word[i]);
    i++;
}
```

```
// the easy way
// using printf with %s
char word[] = "hello";
printf("%s", word);
```



char	char	char	char	char	char
‘h’	‘e’	‘l’	‘l’	‘o’	‘\0’
0	1	2	3	4	5

Strings: How do we read them in?

```
char array[MAX_LENGTH];
// Read in the string into array of length MAX_LENGTH
// from standard input - which by default is the terminal
fgets(array, MAX_LENGTH, stdin);
```

Assume **MAX_LENGTH** is 6 and the user types in **hi** then presses **enter** we would get an array like:

char	char	char	char	char	char
‘h’	‘i’	‘\n’	‘\0’	?	?
0	1	2	3	4	5

string.h library functions

Some other useful functions for strings:

`strlen()`	gives us the length of the string excluding the '\0'
`strncpy()`	copy the contents of one string to another
`strcmp()`	compare two strings
`strncat()`	append one string to the end of another (concatenate)
`strchr()`	find the first occurance of a character in a string

Find more here: https://www.tutorialspoint.com/c_standard_library/string_h.htm

String Functions: strncpy strlen

```c
// Declare an array to store a string
char puppy[MAX_LENGTH] = "Boots";

// Copy the string "Finn" into the word array
// strncpy will truncate the string if it is too long
// this is safer than using strcpy which can cause buffer overflow
strncpy(puppy, "Finn", MAX_LENGTH - 1);
puppy[MAX_LENGTH - 1] = '\0';
printf("%s\n", puppy);
// Find string length. It does NOT include '\0' in the length
int len = strlen(puppy);
printf("%s has length %d\n", puppy, len);
```

Coding Time: strings recap

- Recap strings
  - basics.c
  - full_name.c (not covered in lecture but code example provided)

Array of Strings

```
// This array can store 3 strings.
// Each string has max size 5, including ‘\0’
char words[3][5] = {"hat", "cake", "tea"};
```

- You can have an array of strings!
- You can also think of it as a 2D array of characters

“hat”	“cake”	“tea”
0	1	2

	col 0	col 1	col 2	col 3	col 4
row 0	‘h’	‘a’	‘t’	‘\0’	
row 1	‘c’	‘a’	‘k’	‘e’	‘\0’
row 2	‘t’	‘e’	‘a’	‘\0’	

Array of Strings

```
char words[3][5] = {"hat", "cake", "tea"};
// Using 1 index gives us a row/string
// This would print “cake”
printf("%s\n", words[1]);
```

- You can have an array of strings!
- You can also think of it as a 2D array of characters

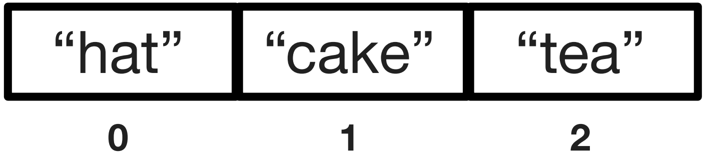
“hat”	“cake”	“tea”
0	1	2

	col 0	col 1	col 2	col 3	col 4
row 0	‘h’	‘a’	‘t’	‘\0’	
row 1	‘c’	‘a’	‘k’	‘e’	‘\0’
row 2	‘t’	‘e’	‘a’	‘\0’	

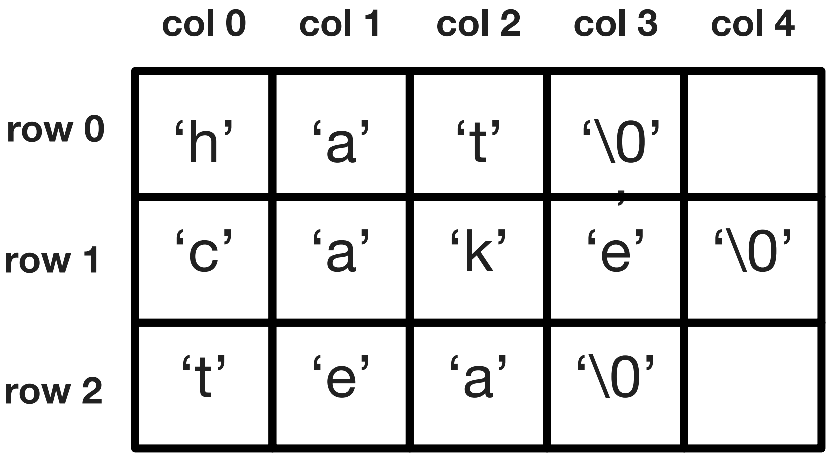
Array of Strings

```
char words[3][5] = {"hat", "cake", "tea"};
// Using 2 indexes gives us a character
// This would print the ‘e’ from “tea”
printf("%c\n",words[2][1]);
```

- You can have an array of strings!
- You can also think of it as a 2D array of characters



“hat”	“cake”	“tea”
0	1	2



	col 0	col 1	col 2	col 3	col 4
row 0	‘h’	‘a’	‘t’	‘\0’	
row 1	‘c’	‘a’	‘k’	‘e’	‘\0’
row 2	‘t’	‘e’	‘a’	‘\0’	

Coding Time Array of Strings

array_of_strings.c

- initialise data
- fgets data
- print out all strings

What are Command Line Arguments?

Command Line Arguments

- So far, we have only given input to our program after we have started running that program (using scanf() or fgets())
- Our main function prototype has always been

  `int main(void);`
- Command line arguments allow us to give inputs to our program at the time that we start running it! E.g.

```
$ dcc prog.c -o prog
$ ./prog argument1 argument2 argument3 argument4
$ ./prog 123 hello
```

Command Line Arguments

- To use command line arguments you need to change your main function prototype to
  `int main(int argc, char *argv[])`
- **`argc`**
  - a counter for how many command line arguments you have (including the program name)
- **`char *argv[]`**
  - an array of the different command line arguments
  - each command line argument is a string (an array of char)

Command Line Arguments

- If we ran our program as follows:

```
$ ./prog 123 dog “hello world”
```

- argc would be equal to 4
- argv would be an array of strings we can visualise as follows:

“./prog”	“123”	“dog”	“hello world”
0	1	2	3

Command Line Arguments

```
int main(int argc, char *argv[]) {
    printf("There are %d command line arguments\n", argc);

    // argv[0] is always the program name
    printf("This program name is %s\n", argv[0]);

    // print out all arguments in the argv array
    for (int i = 0; i < argc; i++) {
        printf("Argument at index %d is %s\n", i, argv[i]);
    }
    return 0;
}
```

Command Line Arguments

```
$ dcc -o command_line_args command_line_args.c
$ ./command_line_args 123 dog "Hello World" COMP1511
This program has 5 command line arguments
This program name is ./command_line_args
Argument at index 0 is ./command_line_args
Argument at index 1 is 123
Argument at index 2 is dog
Argument at index 3 is Hello World
Argument at index 4 is COMP1511
```

Converting Strings to Integers: atoi

- You may want to use your command line arguments to perform calculations, but they are strings!
- There is a function that converts strings to integers:
    - `atoi()` in the standard library: <stdlib.h>
    - E.g. `int x = atoi("952")`
        - Would give us a value of 952 stored in x

Converting Strings to Integers: atoi

```c
int main(int argc, char *argv[]) {
    int sum = 0;
    for (int i = 1; i < argc; i++) {
        sum = sum + atoi(argv[i]);
    }
    printf("%d is the sum of all command line args\n", sum);
    return 0;
}
```

Command Line Arguments

- command_line_args.c
- atoi_demo.c

What are Multi-File Projects?

Multi-File Projects

- Big programs are often spread out over multiple files. There are a number of benefits to this:
    - Improves readability (reduces length of program)
    - You can separate code by subject (modularity)
    - Modules can be written and tested separately
- So far we have already been using the multi-file capability.
    - Every time we #include, we are actually borrowing code from other files
    - We have been only including C standard libraries

Multi-File Projects

- You can also #include your own! (FUN!)
- This allows us to join projects together
- It also allows multiple people to work together on projects out in the real world
- We will also often produce code that we can then use again in other projects
    - that is all that the C standard libraries are - functions that are useful in multiple instances)
- Assignment 2 will be a multi-file assignment.
    - Assignment 1 is not. Do NOT split it up into multiple files

Multi-File Projects

- In a multi file project we might have:
  - (multiple) header files - like the .h files that you have been using from standard libraries already
  - (multiple) implementation files - these are .c files, they implement what is in the corresponding header file.
- a .c file with a main function - this is the entry to our program, we try and have as little code here as possible

Header (.h) Files

- .h files typically contain:
  - function prototypes for the functions that will be implemented in the implementation (.c) file
  - comments that describe how the functions will be used
  - #defines and enums
  - they do not contain executable statements
- .h files give
  - the programmer all the information they need to use the code (a bit like documentation)
  - the compiler the information it needs to do type/syntax checking on the related .c files you `#include` it in

Implementation (.c) Files

- There will be exactly one .c file with a main function
- Other .c files typically contain:
    - Implementations of the functions that you have defined in the corresponding header files
- .c files **`#include`** relevant .h files
    - You use `""` instead of <> to include your own files E.g.
    - **`#include "array_utilities.h"`**

Example: Multi-File C Program

Suppose we have three files:

- header file **`array_utilities.h`**
- implementation file **`array_utilities.c`**
    - **`#include "array_utilities.h"`**
- file with main function **`program.c`**
    - **`#include "array_utilities.h"`**

Compiling Multi-File Programs

- You do **not** compile the **.h** files.
  - They should already be included in the relevant .c files
- You compile .c files together into 1 executable
  - Exactly one of the .c files should have a main function
- E.g.

```
$ dcc -o program program.c utilities.c
$ ./program
```

Feedback Please!

Week 5 Part 2

Your feedback is valuable!

If you have any feedback from today's lecture, please follow the link below or use the QR Code.

Please remember to keep your feedback constructive, so I can action it and improve your learning experience.

https://forms.office.com/r/XasASvSSdY

What did we learn today?

- 2D array of structs with enums coding example
  - mud_and_bones.c
- String recap
  - basics.c (full_name.c code example provided)
- Arrays of strings
  - arrays_of_strings.c
- Command Line Arguments
  - command_line_args.c atoi_demo.c
- Multi-file Programs
  - program.c array_utilities.h array_utilities.c

Reach Out

Content Related Questions:
Forum

Admin related Questions email:
cs1511@unsw.edu.au