COMP1511 - Programming Fundamentals

Week 5 - Lecture 10

What did we cover last lecture?

Debugging

- How to think about different bugs (code errors)
- Some tricks and techniques to remove bugs from our code

Characters

- A new variable type!
- Letters and other symbols

What are we covering today?

Characters

- Continuing characters

Strings

- Words that contain multiple characters

Structs

- Containers that can hold different variable types

Characters in code

```c
#include <stdio.h>

int main (void) {
    // we're using an int to represent a single character
    int character;
    // we can assign a character value using single quotes
    character = 'a';
    // This int representing a character can be used as either
    // a character or a number
    printf("The letter %c has the ASCII value %d.\n", character,
character);
    return 0;
}
```

Note the use of %c in the printf will format the variable as a character

Helpful Functions

`getchar()` is a function that will read a character from input

- Reads a byte from standard input
- Usually returns an int between 0 and 255 (ASCII code of the byte it read)
- Can return a -1 to signify end of input, EOF (which is why we use an int, not a char)
- Sometimes **`getchar`** won't get its input until enter is pressed at the end of a line

`putchar()` is a function that will write a character to output

- Will act very similarly to **`printf("%c", character);`**

Use of getchar() and putchar()

```c
// using getchar() to read a single character from input
int inputChar;
printf("Please enter a character: ");
inputChar = getchar();
printf("The input %c has the ASCII value %d.\n", inputChar, inputChar);

// using putchar() to write a single character to output
putchar(inputChar);
```

Invisible Characters

There are other ASCII codes for “characters” that can’t be seen

- Newline(`\n`) is a character
- Space is a character
- There’s also a special character, EOF (End of File) that signifies that there’s no more input
- EOF has been `#define`d in `stdio.h`, so we use it like a constant
- We can signal the end of input in a Linux terminal by using Ctrl-D

Working with multiple characters

We can read in multiple characters (including space and newline)

This code is worth trying out . . . you get to see that space and newline have ASCII codes!

```
// reading multiple characters in a loop
int readChar;
readChar = getchar();
while (readChar != EOF) {
    printf(
        "I read character: %c, with ASCII code: %d.\n",
        readChar, readChar
    );
    readChar = getchar();
}
```

More Character Functions

<ctype.h> is a useful library that works with characters

- `int isalpha(int c)` will say if the character is a letter
- `int isdigit(int c)` will say if it is a numeral
- `int islower(int c)` will say if a character is a lower case letter
- `int toUpper(int c)` will convert a character to upper case

- There are more! Look up `ctype.h` references or `man` pages for more information

Strings

When we have multiple characters together, we call it a string

- Strings in C are arrays of `char` variables containing ASCII code
- Strings are like words (or sentences), while chars are single letters

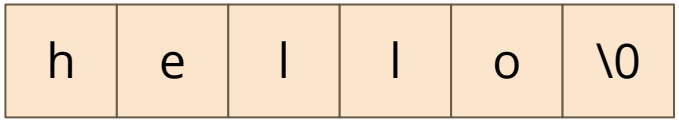
- Strings have a helping element at the end, a character: `'\0'`
- It’s often called the 'null terminator' and it is an invisible character
- This helps us know if we’re at the end of the string

Strings in Code

Strings are arrays of type char, but they have a convenient shorthand

```
// a string is an array of characters
char word1[] = {'h','e','l','l','o','\0'};
// but we also have a convenient shorthand
// that feels more like words
char word2[] = "hello";
```

Both of these strings will be created with 6 elements. The letters `h,e,l,l,o` and the null terminator `\0`



Reading and writing strings

fgets(array[], length, stream) is a useful function for reading strings

- It will take up to **length** number of characters
- They will be written into the **array**
- The characters will be taken from a stream
- Our most commonly used stream is called **`stdin`**, “standard input”

- **`stdin`** is our user typing input into the terminal

Reading and writing strings in code

```
// reading and writing lines of text
char line[MAX_LINE_LENGTH];
while (fgets(line, MAX_LINE_LENGTH, stdin) != NULL) {
    fputs(line, stdout);
}
```

- **`fputs(array, stream)`** works very similarly to printf
- It will output the string stored in the array to a stream
- We can use **`stdout`** which is our stream to write to the terminal

Helpful Functions in the String Library

<string.h> has access to some very useful functions

Note that `char *s` is equivalent to `char s[]` as a function input

- `int strlen(char *s)` - return the length of the string (not including \0)
- `strcpy` and `strncpy` - copy the contents of one string into another
- `strcat` and `strncat` - attach one string to the end of another
- `strcmp` and variations - compare two strings
- `strchr` and `strrchr` - find the first or last occurrence of a character
- And more . . .

Whooaaah We're Halfway There …

We're going to use a bit of everything we've seen so far in COMP1511

This program is a word game

- It will read in a string from the user
- It will then read in another string from the user and tell us how many of the letters from the second appear in the first
- This will use if, while, arrays (of characters), functions and pointers

Where will we start?

A simple version to begin with

- Let's read in a line of characters
- Then read in a single character and see whether it's in the line or not

Read in a line of characters (a string)

We can use a nice library function here

- `fgets()` will grab an entire line from standard input
- We can set up a maximum line size as well

```
#define MAX_LINE_LENGTH 100

int main(void) {
    char line[MAX_LINE_LENGTH];
    fgets(line, MAX_LINE_LENGTH, stdin);
```

Read in a single character

Starting simple, we can take a character as input

- `getchar()` will read a single character from standard input
- Remember that we'll be using int as our type for individual characters
- Here we can loop and continually get characters until input ends

```
int inputChar;
inputChar = getchar();
while (inputChar != EOF) {
    inputChar = getchar();
}
```

Break Time

We're roughly halfway through COMP1511

- This time can sometimes be rough
- It's probably the most tired time of the year for a lot of people

- Remember that you only have to take one step at a time
- Your goals might be so far away that you can't think of how to reach them
- But you only have to move a little bit towards them at a time
- And you'll get there eventually!

A Function to find a character in a string

Loop through the string, testing for a character

- We've done this kind of loop before with other types!

```
int testChar(char c, char *line) {
    int charCount = 0;
    int i = 0;
    while (i < MAX_LINE_LENGTH && line[i] != '\0') {
        if (line[i] == c) {
            charCount++;
        }
        i++;
    }
    return charCount;
}
```

Simple functionality… how well is it working?

What tests should we run at this point?

- Look for syntax errors using our compiler (dcc)
- Look for logical errors by testing with different inputs

We might need to add in some extra outputs

- If we're getting strange behaviour, we can confirm our guesses
- We might learn more about what's going on in our program

What are these extra characters?

Maybe we need to check what those characters are

- Some print statements can help here

```
int inputChar;
inputChar = getchar();
while (inputChar != EOF) {
    printf("Main loop running, readChar is %c.\n", inputChar);
    printf("%d\n", testChar(inputChar, line));
    inputChar = getchar();
}
```

Dealing with little issues

We're reading newlines (`\n`) as characters!

- Let's remove the newlines from both our line and our inputs
- We'll use a library function, **`strlen()`** to find the end of a string
- To use **`strlen()`**, we will need the string.h library, which we will include
- We'll then replace the **`\n`** with **`\0`** which will end the string early

Removing newlines

Removing a `\n` at the end of a string:

```
int main(void) {
    char line[MAX_LINE_LENGTH];
    fgets(line, MAX_LINE_LENGTH, stdin);
    int length = strlen(input);
    input[length - 1] = '\0';
```

Ignoring the `\n` while reading input:

```
    inputChar = getchar();
    if (inputChar == '\n') {
        inputChar = getchar();
    }
```

Expanding on the functionality

Our first attempt just checked for single letters

- Now we expand to words!
- Read in another word
- Check every letter in the word for whether it appears in the phrase
- Then report back how many letters matched

Some good reasons to use functions!

- Reading in words is now duplicated
- We can reuse our testChar() function to see if letters match

A function to read a line

This function also removes the `\n` that fgets will give us

```
void readString(char *input) {
    fgets(input, MAX_LINE_LENGTH, stdin);
    int length = strlen(input);
    input[length - 1] = '\0';
}
```

A function to count letters

Counts how many letters from one string appear in the other

This function also uses another function!

```
int numLetterMatches(char *word, char *line) {
    int i = 0;
    int matchCount = 0;
    while (i < MAX_LINE_LENGTH && word[i] != '\0') {
        if (testChar(word[i], line)) {
            matchCount++;
        }
        i++;
    }
    return matchCount;
}
```

A simple word game

What coding concepts have we used there that might come in handy?

- Characters and Strings (note that we'll never need to memorise the ASCII table to work with characters)
- Using libraries and provided functions
- Loops on strings (using the Null Terminator `\0`)
- Writing multiple functions and using functions within functions
- A lot of our basic C concepts like if, while and array indexing

Structs

A new way of collecting variables together

- Structs (short for structures) are a way to create custom variables
- Structs are variables that are made up of other variables
- They are not limited to a single type like arrays
- They are also able to name their variables
- Structs are like the bento box of variable collections

Before we can use a struct…

Structs are like creating our own variable type

- We need to declare this type before any of the functions that use it
- We declare what a struct is called and what the fields (variables) are

```
struct performer {
    char name[MAX_LENGTH];
    char description[MAX_LENGTH];
    int rank;
};
```

Creating a struct variable and accessing its fields

Declaring and populating a struct variable

- Declaring a struct: "**struct *structname variablename;***"
- Use the . to access any of the fields inside the struct by name

```
int main(void) {
    struct performer rm;
    strcpy(rm.name, "Rap Monster");
    strcpy(rm.description, "Leader");
    rm.rank = 1;

    printf("%s's description is: %s.\n", rm.name, rm.description);
}
```

Accessing Structs through pointers

Pointers and structs go together so often that they have a shorthand!

```
struct performer *rapper = &rm;

// knowledge of pointers suggests using this
*rapper.rank = 100;

// but there's another symbol that automatically
// dereferences the pointer and accesses a field
// inside the struct
rapper->rank = 100;
```

Structs as Variables

Structs can be treated as variables

- Yes, this means arrays of structs are possible
- It also means structs can be some of the variables inside other structs
- In general, it means that once you've defined what a struct is, you use it like any other variable

What did we learn today?

Characters and Strings

- Expanding our variables to letters and words
- A code example to show some of the use of strings
- Using libraries to make strings easier

Structs

- Collections of variables of different types