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

Command Line Arguments

- Input at the moment the program starts running

Characters recap

```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()

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

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

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 read_char;
read_char = getchar();
while (read_char != EOF) {
    printf(
        "I read character: %c, with ASCII code: %d.\n",
        read_char, read_char
    );
    read_char = 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
- Strings are like words (or sentences), while chars are single letters

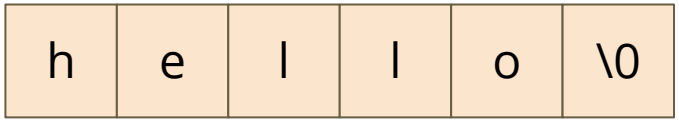
- Strings have a helping element at the end, a character: `'\0'`
- It’s often called the 'null terminator' and it is an invisible character
- This marks the end of the string
- It helps us because we know we won't read any further into the array

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 . . .

Command Line Arguments

Sometimes we want to give information to our program at the moment when we run it

- The **"Command Line"** is where we type in commands into the terminal
- **Arguments** are another word for input parameters

```
$ ./program extra information 1 2 3
```

- This extra text we type after the name of our program can be passed into our program as strings

Main functions that accept arguments

`int main` doesn't have to have `void` input parameters!

```
int main(int argc, char *argv[]) {
}
```

- **argc** will be an "argument count"
- This will be an integer of the number of words that were typed in (including the program name)
- **argv** will be "argument values"
- This will be an array of strings where each string is one of the words

An example of use of arguments and strings

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i = 1;
    printf("Well actually %s says there's no such thing as ", argv[0]);
    while (i < argc) {
        fputs(argv[i], stdout);
        printf(" ");
        i++;
    }
    printf("\n");
}
```

Arguments in argv are always strings

But what if we want to use things like numbers?

- We can read the strings in, but we might want to process them

```
$ ./program extra information 1 2 3
```

- In this example, how do we read `1 2 3` as numbers?
- We can use a library function to convert the strings to integers!
- `strtol()` - "string to long integer" is from the `stdlib.h`

Code for transforming strings to ints

Adding together the command line arguments

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int total = 0;

    int i = 1;
    while (i < argc) {
        total += strtol(argv[i], NULL, 10);
        i++;
    }
    printf("Total is %d.\n", total);

}
```

Break Time

We're roughly halfway through COMP1511

- This time can sometimes be rough
- Sometimes, we're just holding on until the end of the term

- 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!

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 rhyming helper

- It will read in a string from the command line
- It will then read in another string from the user and tell us whether it thinks they might rhyme
- It does this by checking the input string against the last word in the command line and seeing how similar they are
- This will use nearly all the topics we've covered so far in COMP1511

Where will we start?

A simple version to begin with

- Let's read in a string from the command line
- Then read in a single character from standard input and see whether it's in the string or not

Then we complicate things

- We'll try to compare two strings and see if they're similar

Read in strings from the command line

We're expecting these on the command line, so let's check there

- `argc` should tell us how many strings there are

```
int main(int argc, char *argv[]) {
    if (argc <= 1) {
        // there's no extra input on the command line!
        printf("You can't rhyme with nothing!\n");
    } else {
        // continue with the rest of the program
```

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

```
// starting with "input_char = EOF" lets us know later
// whether getchar() replaced it with a character
// or not
int input_char = EOF;
input_char = getchar();
if (input_char != EOF) {
    // we know we've read a character
}
```

A Function to find a character in a string

Looping through a string until the null terminator

```
int check_letter(int letter, char word[]) {
    int found_index = -1;
    int i = 0;

    // The while loop check will loop through
    // until the string is terminated.
    while (word[i] != '\0') {
        if (word[i] == letter) {
            found_index = i;
        }
        i++;
    }
    return found_index;
}
```

We're interested in the last word

How do we know what the last word is?

- `argc` tells us how many words there are!
- So the index of the last word is `argc - 1`
- We can check for the letter in the last word

```
// argv[argc - 1] is the last word of the command line
int found_letter = check_letter(input_char, argv[argc - 1]);
```

Testing a whole word

We could loop `getchar()` to grab multiple characters

- Or we can try another library function that grabs a whole line of text!
- **`fgets()`** will read a line from standard input

```
// read a line of input
char input_line[MAX_LENGTH];
printf("Please enter a word to test for rhyming.\n");
fgets(input_line, MAX_LENGTH, stdin);
```

How well do two words rhyme?

How many letters appear in the other word (not a great test for rhyming)

```
double rhyming_amount(char word1[], char word2[]) {
    // Loop through word1 and check if the letter is in word2
    int match_count = 0;
    int i = strlen(word1) - 1;
    while (i >= 0) {
        int found_letter = check_letter(word1[i], word2);
        if (found_letter >= 0) {
            // found the same letter in the final word
            match_count++;
        }
        i--;
    }
    return (match_count * 1.0)/strlen(word1);
}
```

Using Library Functions

Where does the `strlen()` come from?

- This function will tell us how long a string is
- We need to **`#include <string.h>`** to use it

Are we sure our program is 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

Are there more characters than we intended?

What kind of tests will help us identify the characters?

- Some temporary print statements can help here

```
int check_letter(char letter, char word[]) {
    printf("Checking for %c", letter);
    printf("in word %s.\n", word);
```

```
double rhyming_amount(char word1[], char word2[]) {
    printf("Checking %s", word1);
    printf("against %s.\n", word2);
```

Dealing with little issues

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

- Let's remove the newlines from our **`fgets()`** result
- We'll look for **`\n`** at the end of the string
- We'll then replace the **`\n`** with **`\0`** which will end the string early

Removing a suspected newline

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

```
// read a line of input
    char input_line[MAX_LENGTH];
    printf("Please enter a word to test for rhyming.\n");
    fgets(input_line, MAX_LENGTH, stdin);

// check for a \n at the end of the input and remove it
    int last_letter = strlen(input_line) - 1;
    if (input_line[last_letter] == '\n') {
        input_line[last_letter] = '\0';
    }
```

A simple rhyming helper

What coding concepts have we used here that we want to remember?

- 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

Challenge?

You may have noticed that `rhyming_amount()` loops backwards . . .

- A challenge . . . for bonus Marcs (no actual marks)
- Rhyming amount is a bit simplistic, just checking letter matches
- Can you extend it so that it specifically starts at the end of the words and works backwards and tests the matches for the exact ordering of letters?
- Eg: "light" rhymes with "tonight" because they both end in the same four letters
- There are also more standard library functions that might be able to replace some of our code . . . see if you can discover them

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

Command Line Arguments

- How to take information from the same line that runs the program