COMP1511 - Programming Fundamentals

Term 1, 2019 - Lecture 10
Stream B

What did we cover on Tuesday?

Arrays

- Two dimensional arrays

Functions

- Using libraries

Coco

- An explanation of the game in the first Assignment

What are we covering today?

Assignment 1 and Coco

- How to approach the Assignment
- What needs to be done code-wise
- How the tournaments and marking work

Characters and Strings

- A new variable type!
- Letters and words

Assignment 1

From a practical perspective . . .

- You will write a C program called coco
- It will all be in a single file called coco.c
- Submission is through the give system

The basics

Your program is a Coco player

We will run a **referee program** that will give information to your player and expect a response in return.

Think of the referee as someone typing into a terminal that you will use scanf to read.

Your responses will be text or integers written to the terminal

Let's look at the referee . . .

Open a Terminal on a CSE machine or in VLAB

Run the command `1511 coco_referee -i`

This will allow you to play the game (as a human) against bots just to see how it runs

If you have code that is running, you can run: `1511 coco_referee coco.c`

This will test whether your program is responding correctly to the referee

How to get started

Read the Assignment Spec!

- The spec is available on the class website
- There's an example of a run through of the game

- There's also some **Initial Code**, which you definitely want to start from
- The initial code has a lot of helper comments
- These comments break the problem down into achievable parts

First Things First

How to get the easy marks first

Check the three “modes” you need to run

- Give your name
- Discard Three Cards
- Play a card

Get the program to play legally first. Don’t think about strategy until after you’ve completed that hurdle!

Submit early, submit often

Using “give” will record your submission and back up your work

- It’s much harder to lose your assignment code if we have it!
- If things go bad, you can roll back to previous versions
- You can access your previous versions using our git repository
- The following link is also available in the assignment page:

`https://gitlab.cse.unsw.edu.au/z5555555/19T1-comp1511-ass1/commits/master`

How will your code be tested?

Your program will run once per turn, which means many times per game

- You will be given a single command by the referee
- You will be expected to give a reply and then terminate
- Your choice will be checked by the referee
- Results of games will show whether you've played legally
- They will also give you a command to reproduce any errors

Let's look at a run now with my not-so-functional coco

Using: `1511 coco_referee coco.c`

How will the Tournament and Marking run?

Automated Tournament

- From the 25th through to the due date (7th April)
- We will be running many automated games between submitted players
- You will be able to see how your player is going against other players
- Some of the subject staff will play also

- Your ranking does **not** determine your marks, it's just for fun!
- Your program's capabilities will determine your marks
- The Tournament is there to give you regular feedback on improvements

Marking

How do you earn marks in this assignment?

- **Pass**
  - Code runs without errors
  - Player replies in the right format to the referee commands
  - Player sometimes returns legal plays
  - A serious attempt has been made at the assignment
- **Credit**
  - Player always plays legally
  - Code is reasonably readable

Marking Continued

- **Distinction**
  - There is strategy in the way the Player plays
  - Code is easy to understand and readable
- **High Distinction**
  - Player implements very effective strategies
  - Player shows these strategies against a varied set of opponents
  - Code is perfectly explained and elegant to read

Free Marks!!!

Yep . . . get them right here!

Make your code understandable and readable!

- Follow the Style Guide
- This means correct indentation and consistent use of bracketing
- Use variable names that are understandable to a reader
- Have clear comments explaining your intentions (even if the code is not functional)
- Structure your code file so that different sections are clear
- Use functions to separate repetitive code

Questions?

Feel free to ask any questions now!

- Help Sessions will have Coco decks for practice games

Break Time

Assignment 1

- Tournament starts on the 25th March for anyone who has submitted
- Final due date is April 7th

Characters

We've only used ints and doubles so far

- We have a new type called **char**
- Characters are what we think of as letters, like 'a', 'b', 'c' etc
- They can also represent numbers, like '0', '1','2' etc
- They are actually **8 bit** integers!
- We use them as characters, but they're actually encoded numbers
- ASCII (American Standard Code for Information Interchange)
- We will not be using **char** for individual characters, but we will in arrays

ASCII and Characters as numbers

We make use of ASCII, but we don’t need to memorise it

- ASCII specifically uses values 0-127 and encodes:
  - Upper and Lower case English letters
  - Digits 0-9
  - Punctuation symbols
  - Space and Newline
  - And more . . .
- It’s not necessary to memorise ASCII, rather it’s important to remember that characters can be treated like numbers sometimes

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 int as a character

Helpful Functions

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

- Reads a byte from standard input
- Returns an int between 0 and 255 (ASCII code of the byte it read)
- Sometimes getchar won't get its input until a newline is entered

`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 called EOF (End of File) that signifies that there’s no more input
- EOF has been #defined in stdio.h, so we use it like a constant
- We can signify 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 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 basically words, while chars are letters

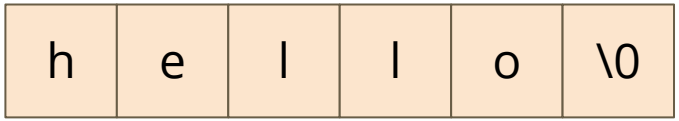
- Strings have a helping element at the end, a 0
- We write it as \0 and it’s often called the null terminator
- This helps us know if we’re at the end of the string

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'};
// but we also have a convenient shorthand
// that feels more like words
char word2[] = "hello";
```

Both of these strings will have 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
- We also have **`stdout`** which is our stream to write to 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 line to the standard output

Helpful Functions in the String Library

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

Note that char* s is equivalent to char s[]

- **`int strlen(char* s)`** - return the length of the string (not including \0)
- **`strcpy`** and **`strncopy`** - 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 . . .

Let’s make a String Program

This is called “Copycat”

- The program will write a line of text
- The user will attempt to write the same line back
- The program will tell them if they were correct

Copycat

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

#define MAX_LENGTH 80

int main (void) {
    char line[] = "Marc is super awesome!";

    // write output to user
    printf("Repeat after me:\n");
    fputs(line, stdout);
    putchar('\n');

    // read user input
    char input[MAX_LENGTH];
    fgets(input, MAX_LENGTH, stdin);
```

Copycat continued

```
    // compare
    if (strcmp(line, input)) {
        printf("You got it! That was 100%% accurate and correct.\n");
    } else {
        printf("Nope . . .\n");
    }
}
```

Something's not quite right . . .

- Check the actual output of strcmp . . . it's not performing as we expect
- When I type in a line, it's not registering as exactly the same?
- Is there something invisible causing the problem?

Copycat fixes

```
// read user input
char input[MAX_LENGTH];
fgets(input, MAX_LENGTH, stdin);

// input will have a \n at the end
// replacing it with \0 will end the string at the last letter
int length = strlen(input);
input[length - 1] = '\0';

// compare
if (strcmp(line, input) == 0) {
    printf("You got it! That was 100%% accurate and correct.\n");
} else {
    printf("Nope . . .\n");
}
```

What did we learn today?

Assignment 1

- What you will be doing as a programmer
- How the program should perform
- How you will be assessed

Characters and Strings

- Using letters and words
- A lot of new functions and terminology!
- We'll practice these more in Tutorials and Labs