Programming Concepts

Assignment 1: Console Life

Date: 22/11/2017

Name: Joshua Todd

Module: SDFD102

Abstract

An explanation with examples of testing into the workability of the functions and programs created to emulate the Game of Life, create random coloured blocks on the screen, create a controllable moving block and a prime number calculator. Also involves extra planning documents, research, explanations of different errors and improvements on possible problems.

[**Task 1**](#_vs9bm364usha) **3**

[Menu With 3 Programs, What Does It All Do?](#_1ppntjw9898h) 3

[main();](#_wgsktljl608y) 3

[clearScreen();](#_ejh8swsfjlqs) 3

[gotoxy();](#_trwbztpkvxg8) 4

[textcolor();](#_6bldsn8quv2s) 4

[menu();](#_nbpkius3qp4y) 4

[movingBlock();](#_1s2xb6jjrkws) 5

[randomColours();](#_87ac5obmlhc3) 7

[gameOfLife();](#_v6io24pdcfps) 8

[Flowchart for the Menu System](#_uusb62q4qct6) 12

[Testing](#_wsz9c9gtucfw) 13

[Instructions](#_aj4n49x22fn8) 14

[Possible Improvements and Problems](#_ywj0syqeka9a) 15

[Syntax and Semantic Errors](#_z1eoz7bc8p8q) 16

[Syntax Errors](#_p4s4wjlm1iuo) 16

[Semantic Errors](#_j7dsu6wwjb2) 17

[**Task 2**](#_tmyvjdr5q542) **18**

[Flowchart to Find a Prime Number](#_na97c8haka93) 18

[First 100 Prime Numbers](#_gizbm19pq4h6) 19

[Test Plan](#_ht1q9g5tkqs7) 20

[Test Items](#_fezjo21lwm12) 20

[Features to be Tested](#_or6wypf1m9wg) 20

[Resources](#_t4lys7xzivy4) 20

[Approach](#_bmf5yeo53beu) 20

[Pass Criteria](#_v4gdp3euhfiv) 20

[Optimisation](#_owmlr1xksh1) 21

[Optimisation 1](#_2u09y6tqv75m) 21

[Optimisation 2](#_o2me8zj5jcg) 21

[Optimisation 3](#_ndotuyjb9jeu) 21

[Different Methods of Finding Prime Numbers](#_17ga9fr806rw) 22

[Eratosthenes](#_1yozt1plzh) 22

[Prime Number Theorem](#_ozuq83ffrr7z) 22

[References](#_duwtd3a0eh3y) 22

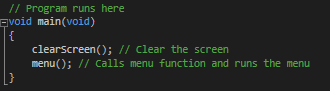
# Task 1

## Menu With 3 Programs, What Does It All Do?

(Source code in separate file, too big for google doc.)

### main();

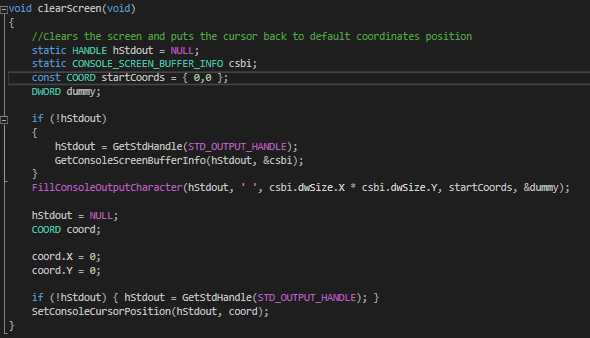
The main function of the program contains only 2 lines, it calls the clear screen function and the menu function.



These functions will be explained below.

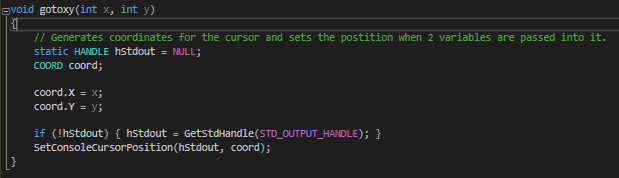
### clearScreen();

The clear screen function essentially resets the console screen and wipes any text already on there, it also resets the coordinates of the cursor to 0,0.



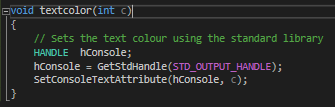
### gotoxy();

The gotoxy function takes 2 integers into its parameters and puts the cursor at the coordinates of those 2 integers.



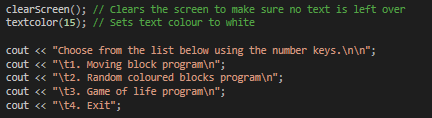
### textcolor();

The textcolor function allows you to change the colour of any text by passing it a single integer.



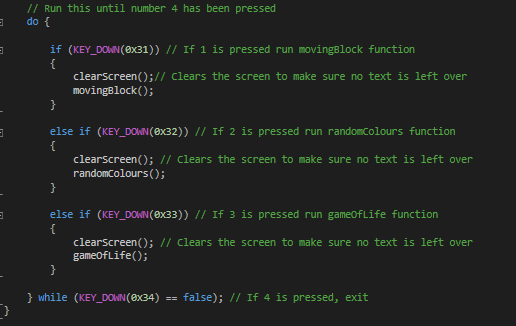
### menu();

This function creates the main menu of the program.



First the screen clears by calling the clearScreen function, secondly the text color is changed to white; the integer 15 will make text color white. This applies to all outputted text below it within the function.

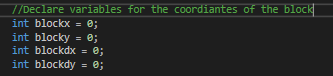
The ‘cout’s’ contain ‘\n’ to put the next line of text on a new line, making readability easy for the user.



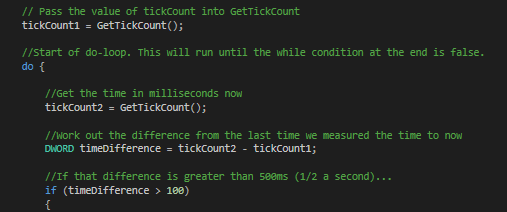
Above, the do loop runs until the number 4 key is pressed, when pressed the code will exit. Within the do loop there are 3 if statements that check whether the user has pressed 1, 2 or 3. Depending on what number is pressed it will call a specific function.

### movingBlock();

This function contains the code for the Moving Block program.

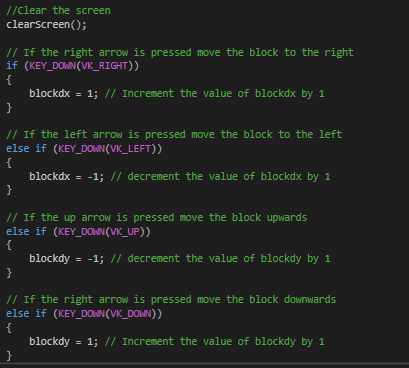


First the variables for the coordinates of the random blocks are declared.



It starts a do loop and a tick count within an if statement to run every 10th of a second.

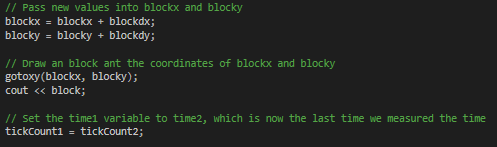
Within the if statement it checks for user inputs from the arrow keys on the keyboard and moves the block in the corresponding direction.



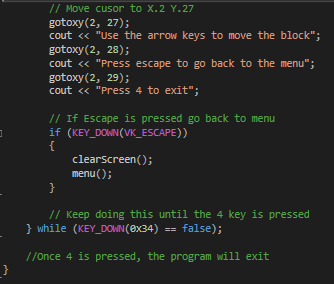
If no inputs are detected then the coordinates of the block stay the same.



If an arrow key is pressed, it passes the value of the modified variables to the existing value of a second variable and makes that the new value. It then outputs the block at the coordinates of the second variables (X and Y) and sets the first tickCount to the value of the second tickCount.

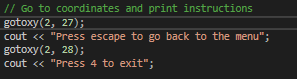


Finally it prints out the instructions at the bottom of the program to tell the user how to interact with the program. If escape key is pressed it will go back to the main menu and if 4 is pressed it will exit the program.



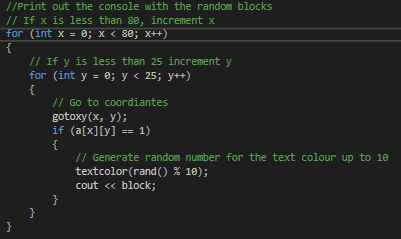
### randomColours();

This function runs the Randoms Colours program, first of all it prints out instructions.

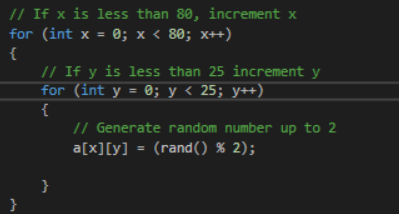


Then it starts a do loop which runs until 4 is pressed and closes. Within the loop are 2 for statements and an if statement.

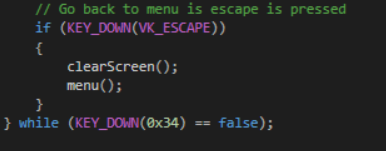
While x is less than 80 and y is less than 25 it goes to the coordinates of x and y. If the array a corresponds to 1 it prints out a series of random blocks and assigns the colour of the text randomly from 1 to 10.



The second for loop is similar and randomises where the next block will be printed before it is printed.

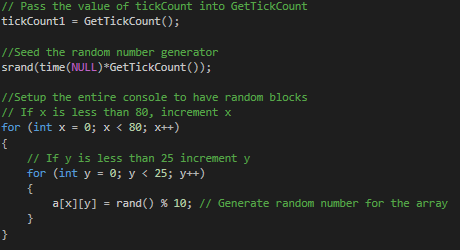


The last part does the same as the moving block function above.

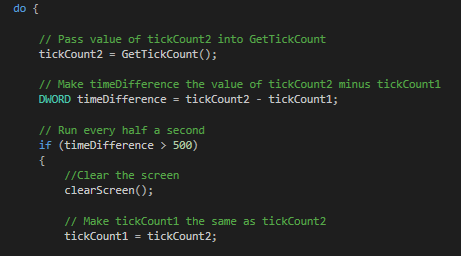


### gameOfLife();

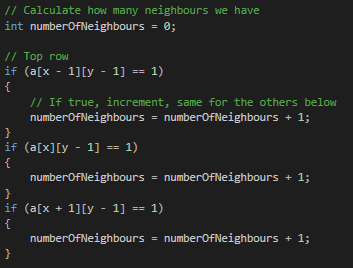
Firstly this function generates a tick to force the program to run at a set speed, it then seed the random number generator. After it randomly generates an array within x80, y25.



It then runs a do loop and uses the difference between tickCount1 and tickCount2 to figure out the length of a second, it runs every half second and clears the console and resets the tick.



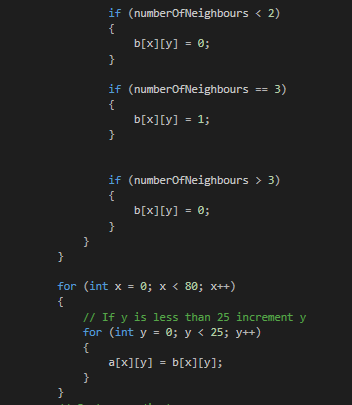
After printing out a block in the same way as the previous 2 functions, the program uses if statements to calculate the number of neighbours each block has.



The conditions for the if statements: if a[x-1][y-1] is equal to 1 add 1 to number of neighbours, it does this for the 3 top row locations and the same for the middle and bottom row essentially completing a scan of a 3x3 grid around the location of a block and checks which spots are populated or not.

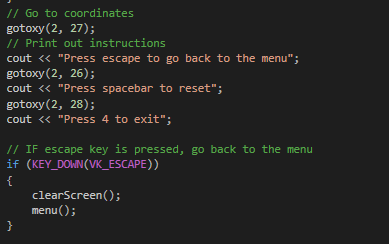
With this information it looks at the value of numberOfNeighbours and decides using if statements if it should change the value of the b array.

The point of a second array (b array) is to be able to put all the updated information into b once it has decided where to put all the neighbours then put all that information back into a array at once to print out all the blocks at once, instead of one by one.

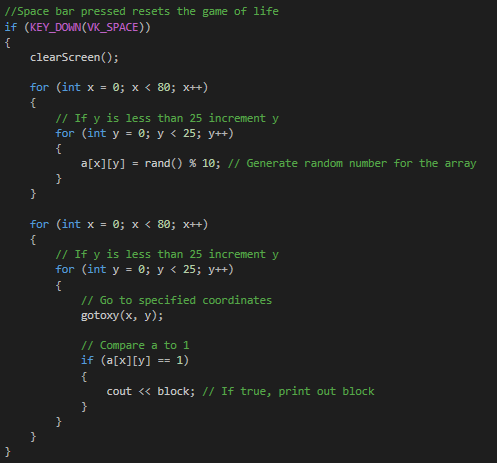


It does that in the for loop above.

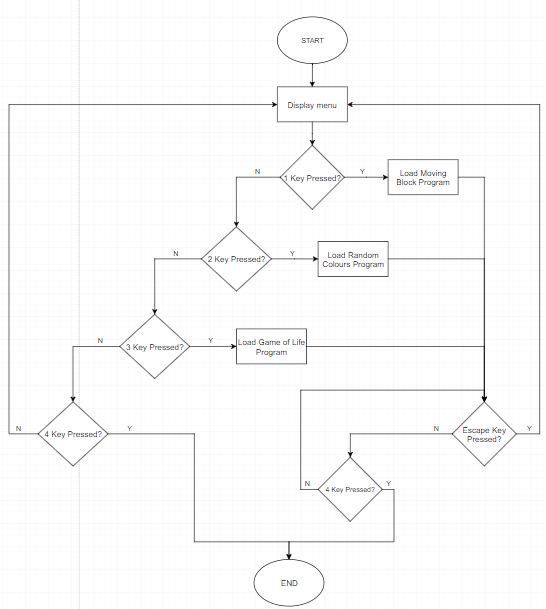
Furthermore it prints out the instructions at a specific location using the gotoxy function again.



Finally the code at the bottom of the function resets the console and re-prints out the random blocks if the spacebar is pressed.

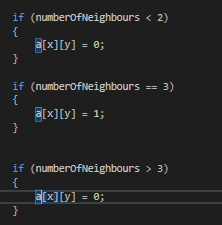


## Flowchart for the Menu System



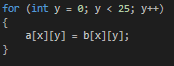
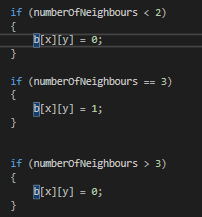
## Testing

On the game of life program, previously I had thought it was complete, however all the blocks were updating at different times meaning the simulation was incorrect.

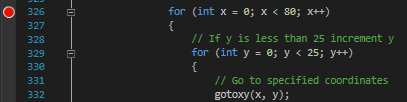


This was how I updated the array, it produced a similar looking game of life but was not correct.

To correct this problem I had to put the updated information into a temporary array and then pass that information into array ‘a’ to print out all the updated blocks at once.



To test for problems like this or other problems in code that are hard to figure out, you can use breakpoints to stop the code running at certain points, this lets you slowly cycle through lines of code to find out the problem.



## Instructions

**Menu Keys:**

1 Key = Start the moving block program

2 Key = Start the random colours program

3 Key = Start the game of life program

4 Key = Exits

**Moving Block Program keys:**

Up Arrow Key = Moves the block up

Down Arrow Key = Moves the block down

Left Arrow Key = Moves the block left

RIght Arrow Key = Moves the block right

Escape Key = Back to the menu

4 Key = Exits

**Random Colours Program keys:**

Escape Key= Back to the menu

4 Key = Exits

**Game of Life Keys:**

Space Bar Key = Resets the game of life

Escape Key= Back to the menu

4 Key = Exits

## Possible Improvements and Problems

In my opinion i think that pressing 4 to exit the code is a bit strange, I think it should be the escape key, however that key is used to go back to the previous menu as specified in the task instructions.

The code could be optimised in some way, the source code is currently a very big block of text, for speed purposes only it could be improved but I don’t think it is necessary as speed is not the aim of this task.

A problem I have encountered is when you choose a program, Game of Life for example and then press the key for another program (1 or 2), when you press escape to exit out the menu, instead of going to the menu it will straight away run a different program depending on what key you pressed.

Currently when the console runs it does not tab you out of other programs on your pc, so when pressing buttons it can affect anything else you have open.

In the moving block program the block will move diagonally if you press 2 keys in quick succession, I think this is happens when 2 buttons are pressed within 1 tick of the program.

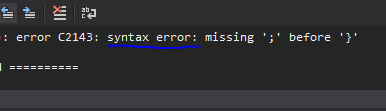
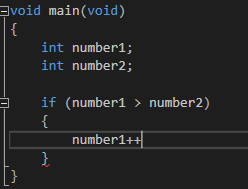
The block is also able to go off the boundary of the array and will just display the block in the top left corner until the coordinates of the array come back into 80 by 25. These are all things that can be improved in the program.

## Syntax and Semantic Errors

### Syntax Errors

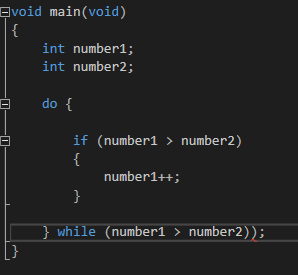
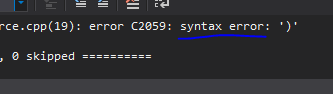
A syntax error is a violation of the syntax of any set of rules given to any language.

For example; ‘we am very happy’ is an example of an error of English syntax even. Even though we know what that sentence means it is much more strict in C++ programming, an error such as forgetting to put a semicolon at the end of a line will make the compiler throw a syntax error.



To solve this error all you would have to do is put a semicolon at the end of number1++.

Another example is an imbalance of parenthesis.

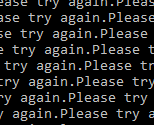


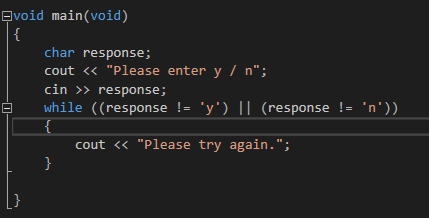
To solve this error the parenthesis with the red line underneath would need to be deleted to make the balance of parenthesis correct.

### Semantic Errors

Semantic errors occur after syntax errors, when the code is understood by the compiler however it does not do as the user intended.

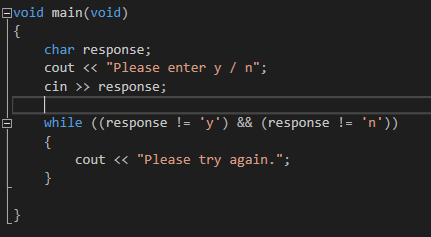
The example below shows how the code creates an unintended infinite loop.





The code compiles, however the expression (response != ’y’) || response != ’n’) is always going to be true no matter how many times the user inputs y or n, the code will never escape the while loop.

To correct this the OR operator can be changed to the AND operator like so:

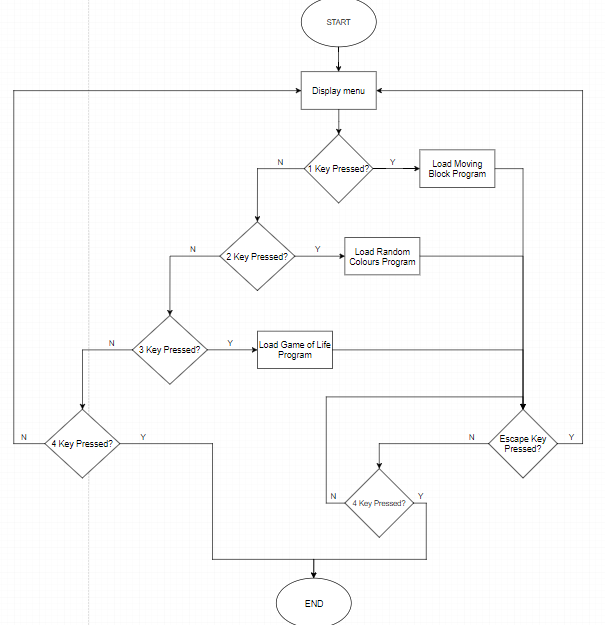


Now pressing y or n will exit the while loop, however it will still loop if another key is pressed.

In short, Syntax errors are errors thrown by the compiler because it cannot understand the code it has been given and semantic errors are unintended errors that the compiler will still execute anyway.

# Task 2

## Flowchart to Find a Prime Number



## First 100 Prime Numbers

The code was tested against the first 100 prime numbers as a good indication that the solution works correctly.

*“A prime number is a positive integer that has exactly two positive integer factors, 1 and itself. For example, if we list the factors of 28, we have 1, 2, 4, 7, 14, and 28. That's six factors. If we list the factors of 29, we only have 1 and 29. That's two factors. So we say that 29 is a prime number, but 28 isn't.*

*Here are the first few prime numbers:*

*2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97”*

*(Dr Math.)*



The first 100 prime numbers match up to other sources of the first 100 prime numbers, from this we know that the calculation is correct for prime numbers up to 10000.

## Test Plan

### Test Items

* primeNumbers.cpp

### Features to be Tested

* For loop for finding out prime numbers
* Timer to tell the user the time taken to find the prime numbers
* The output of data to the user
* Do loop to show all prime numbers

### Resources

* Flow chart
* Data for first 100 prime numbers
* Tick count created in task 1

### Approach

* Create variables
* Start a tick count
* Create for loop to calculate prime numbers
* Output data to user
* Space key shows prime numbers
* Optimise code

### Pass Criteria

* The program is to find all the primes in the range listed above and store them in memory.
* The total time required to perform the check is to be measured by the program.
* Once the calculation is complete the total number of prime numbers found within the range is to be shown.
* Code is optimised

## Optimisation

There are a few ways the code was optimised to get a faster time

### Optimisation 1

First of all the code printed out every number and said whether or not the number was prime.





Printing is a very slow process, we don’t need to show all the numbers and whether or not they are prime, only all the prime numbers found after is the user would like to see them.



### Optimisation 2

First of all the for loop had the condition:



I realised that the code does not need to check for even numbers as all prime numbers are odd numbers. Instead of incrementing by 1 we can increment by 2 from the number 3 meaning it will only check odd numbers.





### Optimisation 3

Using fmod to calculate a remainder is a slow process.





Instead this was changed to use modulo, which deemed mush faster.

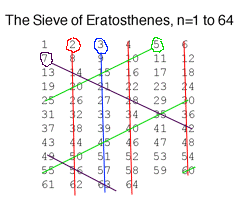




**The amount of numbers to check for primes was changed to 100,000 to meet task requirements.**

## Different Methods of Finding Prime Numbers

### Eratosthenes

In the 3rd century B.C, Eratosthenes developed a method of finding prime numbers. Here is an example for the numbers 1 to 64.

The square root of 64 is 8 so first we find all the prime numbers up till 8. Draw a grid of all the numbers from one to 64. Circle all the prime numbers found up till 8 and all multiples of these numbers we can cross out, the numbers left in the grid that are not crossed out will be prime numbers.

The method is simple, however finding out big prime numbers is very time consuming with this method.

**[Figure 1]** - Grid from The Math Forum

### Prime Number Theorem

*“In 1896 Charles de la Vallee-Poussin and Jacques Hadamard proved the Prime Number Theorem, which states:*

*Let Pr(x) be the number of prime numbers less than x. Then the ratio of Pr(x) to (x/ln(x)) approaches 1 as x grows without bound.”(The Math Forum).*

This means if n is a prime number the distance to the next prime number is approximately ln(n).

## References

Anonymous. ‘*Prime Numbers*‘ (Accessed at: <http://mathforum.org/dr.math/faq/faq.prime.num.html>) [Last Accessed: 22 November 2017]

Anonymous. [Figure 1]. ‘*How do you find Prime Numbers?*‘ (Accessed at:

<http://mathforum.org/isaac/problems/prime2.html>) [Last Accessed: 22 November 2017]

Anonymous. ‘*How do you find Prime Numbers?*‘ (Accessed at: <http://mathforum.org/isaac/problems/prime2.html>) [Last Accessed: 22 November 2017]