Introduction to Programming Glossary

By Reuben Wilson (9988289)

Programming Overview

By using functional decomposition in my *TestUserInput.pas* program, I was able to break the program down into all the blocks, or functions that were required for the program to work properly. By constructing these blocks first and knowing they work, I didn't have to worry about the code, I could just call upon them when needed. Below is an example of functional decomposition in use:

```
function ReadInteger(prompt: String): Integer;
var
    userInput: String;
begin
    userInput := ReadString(prompt);

    while not TryStrToInt(userInput, result) do
    begin
        WriteLn('Please enter a whole number: ');
        userInput := ReadString(prompt);
    end;
end;

function ReadIntegerRange(prompt: String; min, max: Integer): Integer;
begin
    result := ReadInteger(prompt);
    while (result < min) or (result > max) do
    begin
        WriteLn('Please enter a number between ', min, ' and ', max,',');
        result := ReadInteger(prompt);
    end;
end;
```

As you can see, the function *ReadInteger* is its own block, the function *ReadIntegerRange* calls upon *ReadInteger* to calculate its result. Below is what *ReadIntegerRange* would look like if it had to implement all of the functionality required by it.

```
function ReadIntegerRange(prompt: String; min, max: Integer): Integer;
var
    userInput: String;
begin
    userInput := ReadString(prompt);

    while not TryStrToInt(userInput, result) do
    begin
        WriteLn('Please enter a whole number: ');
        userInput := ReadString(prompt);
    end;
    result := userInput;
    while (result < min) or (result > max) do
    begin
        WriteLn('Please enter a number between ', min, ' and ', max,',');
        result := ReadInteger(prompt);
    end;
end;
```

Good Programming Practices:

The program, which was really quite simple, was instructed to draw four bikes on the screen. The title really did it no justice, not only was it extremely messy, it was extremely difficult to follow the flow of the code.

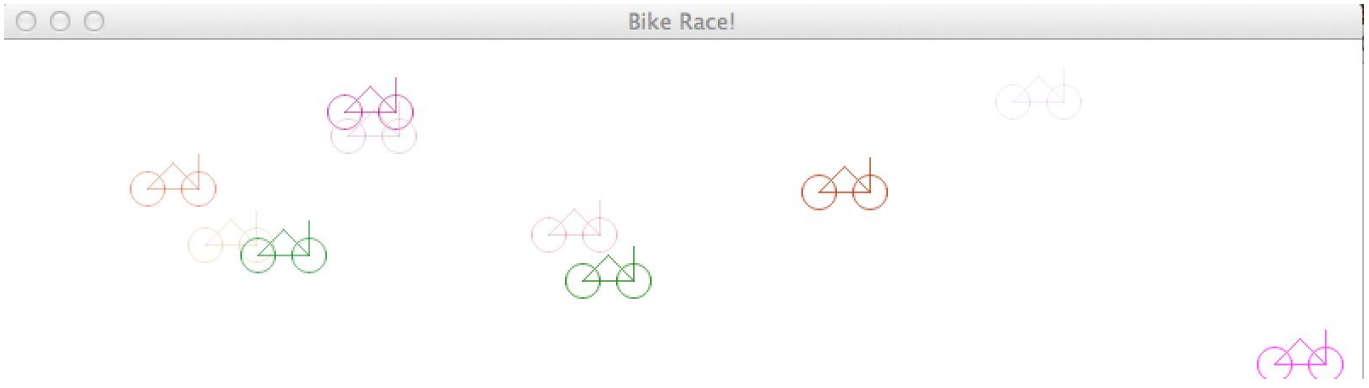
The factors that really contributed to the overall difficulty of understanding were poor indentation practices, ridiculously vague variable names and enough unnecessary repetition to drive anyone insane.

To make things more clear, I removed the repetition of drawing a bike four times over with messy code by implementing a procedure, which handled the execution of displaying a bike to the screen. I changed the names of the variables to make more logical sense and I got rid of two constants, which were of little to no consequence.

I used an easy implementation of coded boundaries to fix the bug, which would frequently occasion the bike off the screen. Take a squiz at this snippet of code:

```
if (bikeX > (ScreenWidth() - BIKE_WIDTH)) then
    begin
        bikeX := (ScreenWidth() - BIKE_WIDTH)
    end;
if (bikeY > (ScreenHeight() - BIKE_HEIGHT)) then
    begin
        bikeY := (ScreenHeight() - BIKE_HEIGHT)
    end;
```

Here's a screenshot of the output of the messy bike program after I'd finished making it more user friendly:



Functional Decomposition

When we approach even the biggest programming problems, we need not be worried about their complexity due to the way we can implement functional decomposition to make a daunting problem a simple, easy to understand task.

By breaking down large problems into smaller problems, or tasks, we can begin to have a better understanding of what it is our program needs to do and how we, as programmers, can implement those needs stress free. These smaller tasks can be the functions and procedures that the program needs to run.

Once all the identified functions and procedures have been pieced together and work, they can be combined to make one program! Think of bricks in a house, we can't construct our house until we have all the bricks needed!

Function Call

Example 1: Assignment statement with function call

This example shows the function *ReadIntegerRange* calculating a value and assigning it to the variable *guess*.

```
guess := ReadIntegerRange('What do you think it is? ', min, max);
```

For an idea of the value that is calculated, refer to the below snippet:

```
function ReadIntegerRange(prompt: String; min, max: Integer): Integer;
begin
    result := ReadInteger(prompt);
    while (result < min) or (result > max) do
    begin
        WriteLn('That is clearly not a number  between ', min, ' and ', max,'. Bitch.');
        result := ReadInteger(prompt);
    end;
end;
```

Example 2: Function Call to get parameter value

This example shows the *SwinGame* procedure *DrawRectangle* being called with the following values passed:

The colour is *circleColor*, the X & Y positions are being determined by the function *Random*. The width and height of the rectangle are also being determined by the *Random* function. So, this is indeed an example of a procedure being passed functions as parameters.

```
DrawRectangle(circleColor, Random(ScreenWidth()), Random(ScreenHeight()), Random(10) + 1, Random(10) +1);
```

Example 3: Function Call in condition

This small example shows the result of *ReadIntegerRange* being assigned the value returned by the function *ReadIntegerRange*. That value is then used in a *while* loop as shown in the code below.

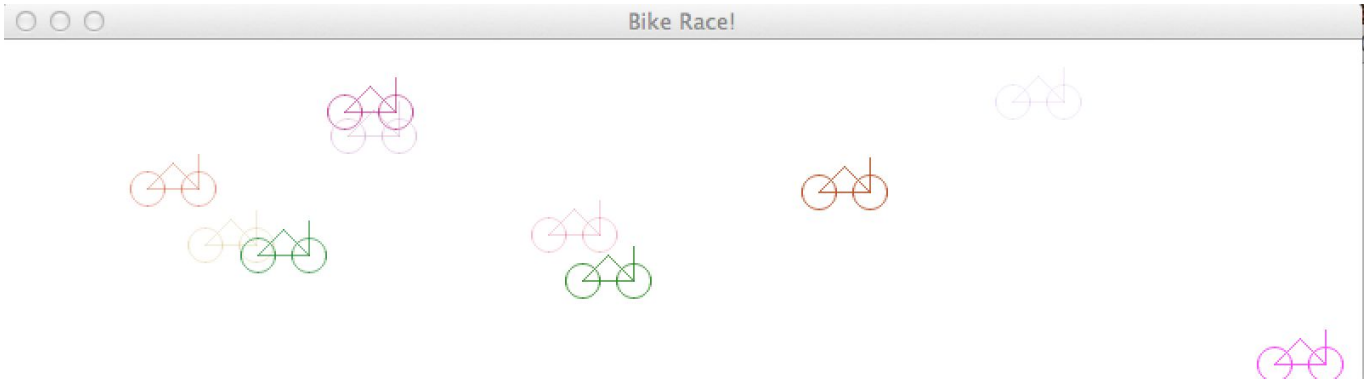
```
function ReadIntegerRange(prompt: String; min, max: Integer): Integer;
begin
    result := ReadInteger(prompt);
    while (result < min) or (result > max) do
    begin
        WriteLn('That is clearly not a number  between ', min, ' and ', max,'. Bitch.');
        result := ReadInteger(prompt);
    end;
end;
```

Core Exercise 2 – User Input Functions:

Here is my sequence diagram for the implemented *ReadDouble* and *ReadDoubleRange.*
Please find the code for the program *TestUserInput2.pas* and the unit *UserInput.pas* below.

Core Exercise 3 – Messy Program:

Here is a screenshot of my redesigned program:



Please find the code for this program below.

Extension Exercise 1 – Guess That Number:

Here are some screenshots of my *GuessThatNumberPro.pas* program running:

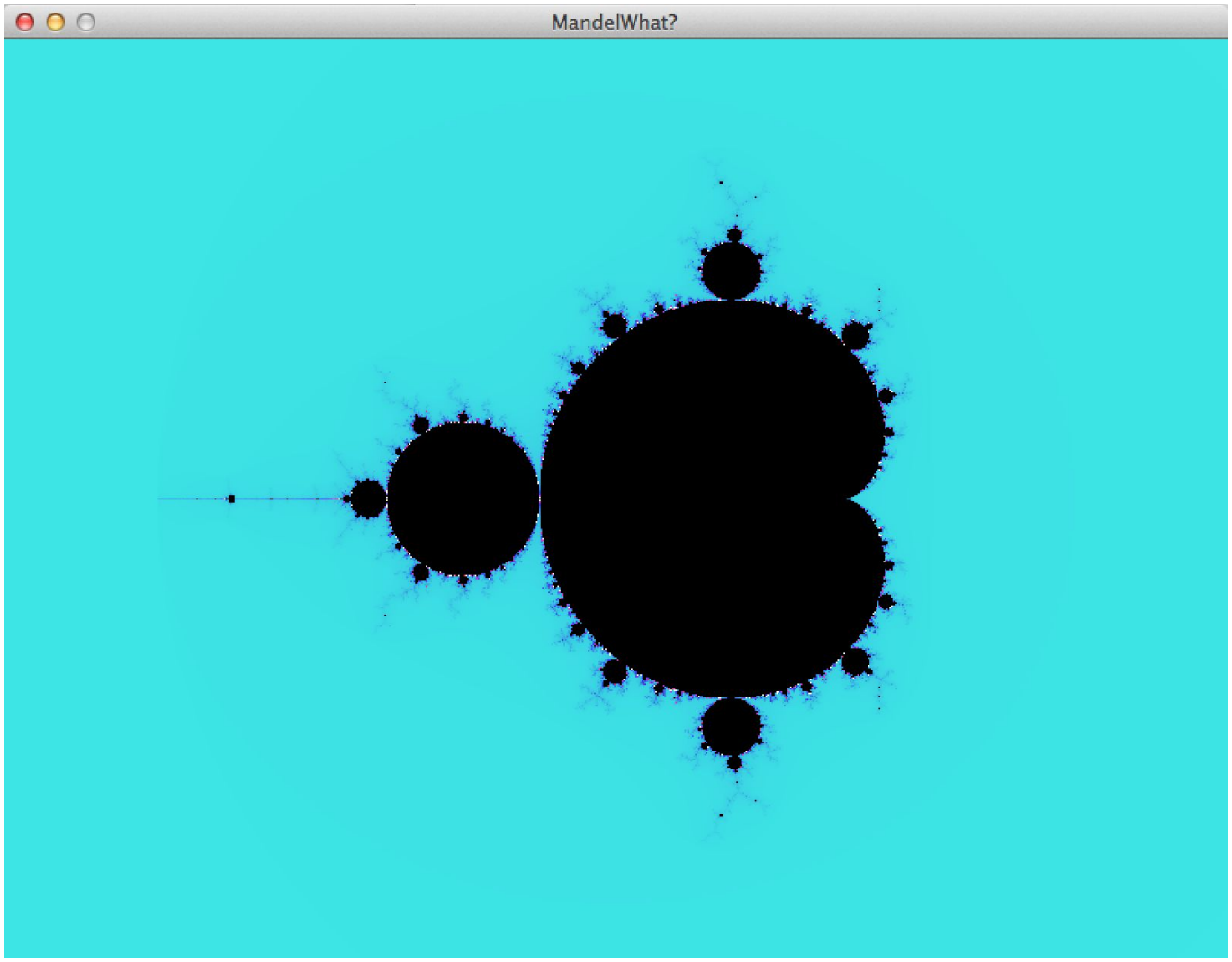
```
Welcome to Guess that number!
Select an option ->
 - [E]asy Mode
 - [H]ard Mode!
 - [Q]uit
 - [M]aybe
Option ->
e

I'm thinking of a number between 1 and 100

Guess # 1
What do you think it is?  50
The number is larger than that.

Guess # 2
What do you think it is?  75
The number is less than that

Guess # 3
What do you think it is?  62
The number is larger than that.

Guess # 4
What do you think it is?  69
The number is less than that

Guess # 5
What do you think it is?  66
The number is less than that

Guess # 6
What do you think it is?  64
The number is larger than that.

Guess # 7
What do you think it is?  65
Good guess mother home_dawg!
Woohoooo!


Do you wish to play again
Select an option ->
 - [E]asy Mode
 - [H]ard Mode!
 - [Q]uit
 - [M]aybe
Option ->
```

```
I'm thinking of a number between 1 and 1000

Guess # 1
What do you think it is?  100
The number is larger than that.

Guess # 2
What do you think it is?  500
The number is larger than that.

Guess # 3
What do you think it is?  23
The number is larger than that.

Guess # 4
What do you think it is?  5
The number is larger than that.

Guess # 5
What do you think it is?  3
The number is larger than that.

Guess # 6
What do you think it is?  5
The number is larger than that.

Guess # 7
What do you think it is?  3
The number is larger than that.

You ran out of guesses homie!
The answer was 588
Maybe Try again!

Do you wish to play again
Select an option ->
 - [E]asy Mode
 - [H]ard Mode!
 - [Q]uit
 - [M]aybe
Option ->
```

Please find the code for this program attached.

Extension Exercise 2 – Mandelbrot:

Here are some screenshots of my *Mandrelbrot.pas* program running:



Please find the code for this program attached.