# Introduction to Programming Glossary

*By Reuben Wilson (9988289)*

## Programming Overview

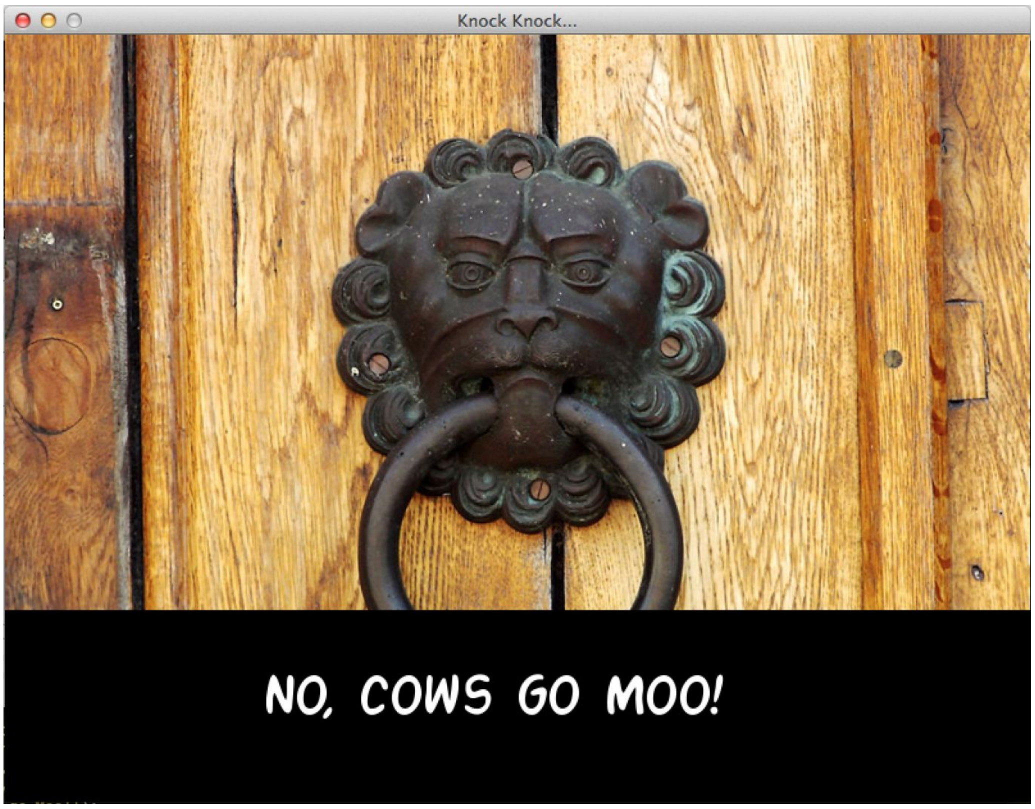
My *Knock Knock* program uses the *SwinGame* Pascal library. It's important that this link is understood because *SwinGame* is responsible for the way that most of the procedures used in this program behave.

*Knock Knock* is a simple program, which consists of a number of procedures in sequence. Most of these procedures use parameters or **data**. Here is an example of how one of the procedures uses defined parameters:

```
procedure PunchLine(text: String);
begin
  // Draws the joke text as White, using the 'joke font' at the position on the screen noted by 200, 500.
  // Show the current screen.
  // Play the punch line laugh.
  // Give the user 2000ms, or 2 seconds to thouroughly enjoy my fantastic humor
  // Then call the procedure ClearAreaForText which as we know prints a black rectangle over the last piece of text.
  DrawText(text, ColorWhite, 'joke font', 200, 500);
  RefreshScreen();
  PlaySoundEffect('laugh');
  Delay(2000);
  ClearAreaForText();
end;
```

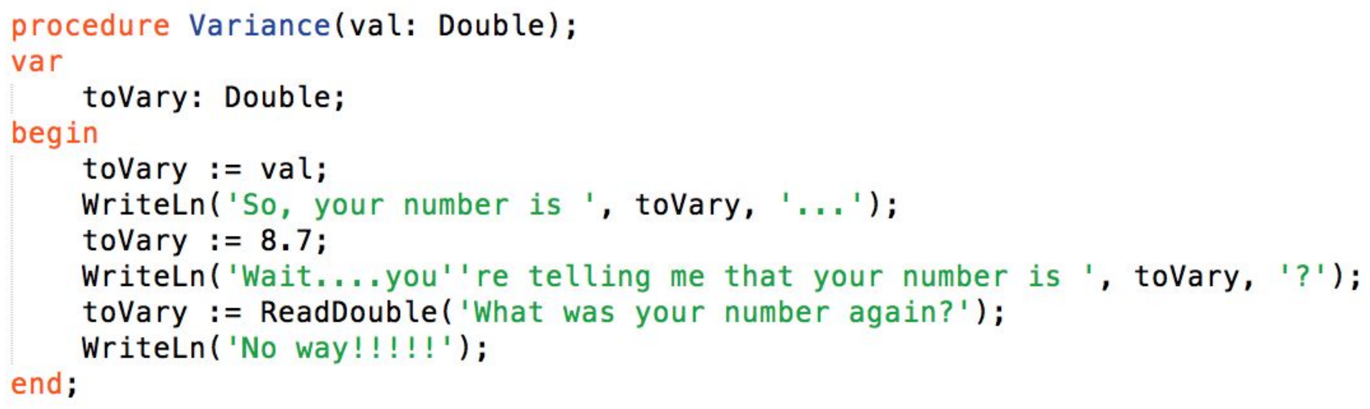
By referring to the above procedure, *PunchLine*, it is clear that it is going to be passed a parameter, in this case a string. By referring to the code comments, an understanding can be established as to how this procedure is going to deal with the passed parameter.

E.g. If the string *'No, cows go Moo!'* was to be passed to *PunchLine*, this is what the result would look like



Variable: A type of artefact, used for storing a single value. The value can be varied, or changed!

Variables are declared and utilised locally in functions and procedures and, based on the definition above; their value can be varied. As an example, consider the procedure *Variance* below:

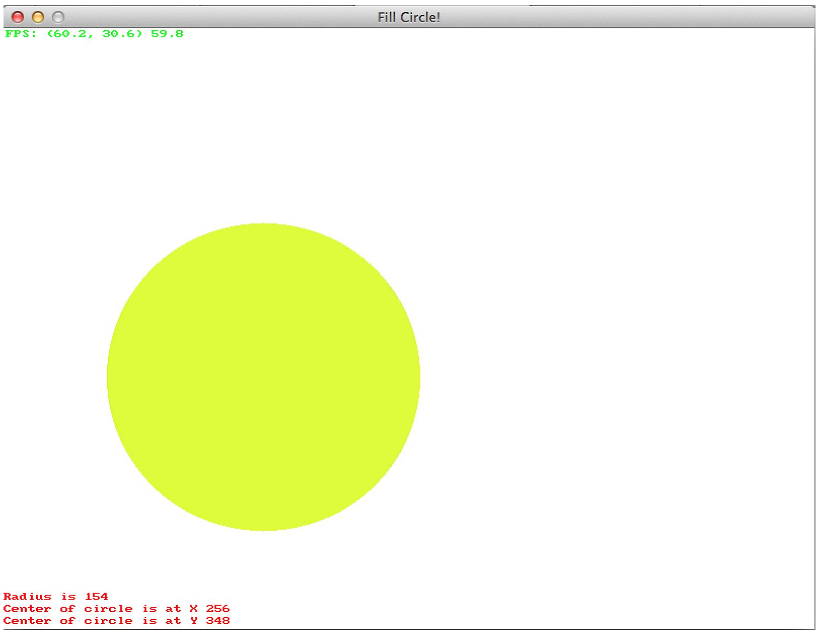


```
procedure Variance(val: Double);
var
    toVary: Double;
begin
    toVary := val;
    WriteLn('So, your number is ', toVary, '...');
    toVary := 8.7;
    WriteLn('Wait....you''re telling me that your number is ', toVary, '?');
    toVary := ReadDouble('What was your number again?');
    WriteLn('No way!!!!!');
end;
```

In this particular procedure, the variable *toVary* is declared to stored a single value of data type double. As depicted, *toVary* will initially be assigned the value of *val*, a parameter that is passed to the procedure. Then, *toVary* is assigned the value 8.7, meaning the value has been varied. Finally, *toVary* is changed once more by assigining to the value returned by calling the function *ReadDouble*. This procedure accurately demonstrates how a variables value may vary.

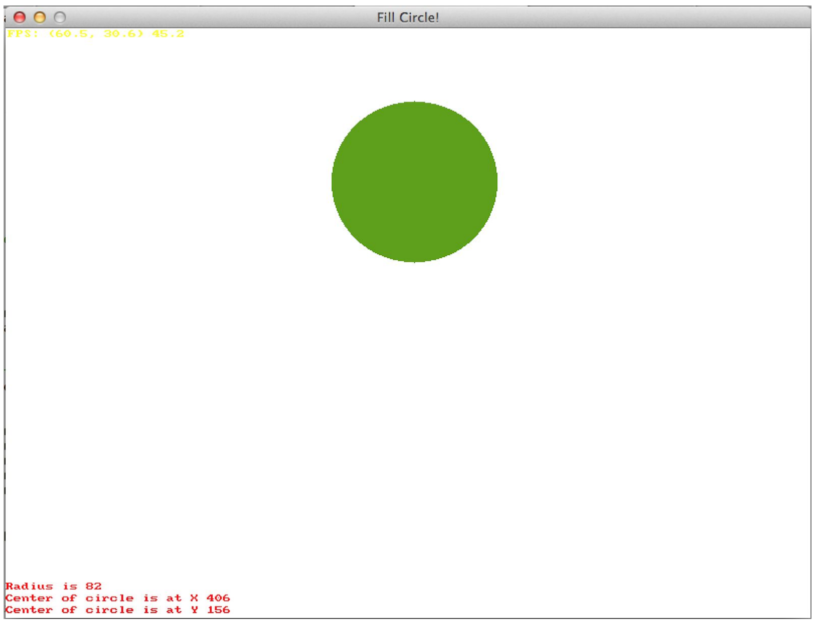
The implementation of control flow makes programs so much more interactive and dynamic. Before being introduced to control flow this semester, all the code that we had written was great in terms of execution and the desired output, but the programs were extremely limited with what they could do. They would only perform tasks in the order in which they were specified, and once the desired output was met, the program would terminate. This seems robotic and impractical because the user can only interact with the program within its specified bounds.

Using control flow in programs drastically improves useability by implementing a system, which makes programs more dynamic and interactive. How is this achieved? By using control flow, you can alter the sequence of execution of code within the program based on conditions. For example, by using control flow, I can draw a circle to the screen; change its size, colour and location, amongst other variables; based entirely on the user's input. Below are some nifty screenshots of a program I made, which does just that:



```
FPS: (60.2, 30.6) 59.8
```

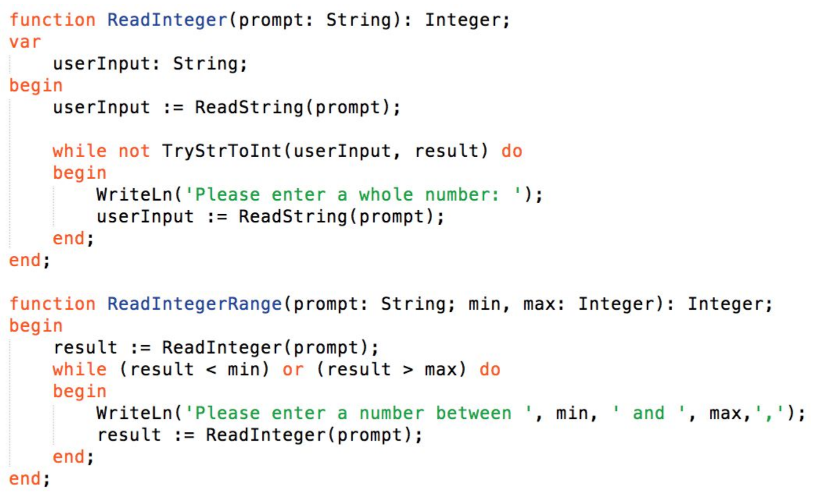
```
Radius is 154
Center of circle is at X 256
Center of circle is at Y 348
```



```
FPS: (60.5, 30.6) 45.2
```

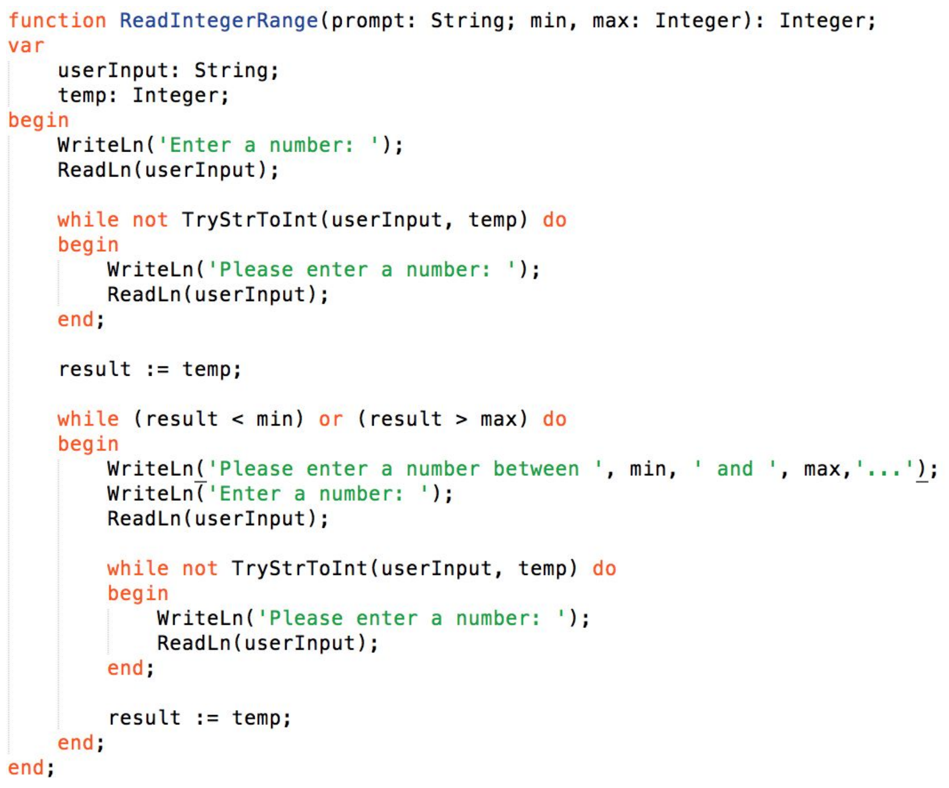
```
Radius is 82
Center of circle is at X 406
Center of circle is at Y 156
```

By using functional decomposition in my *TestUserInput* 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:



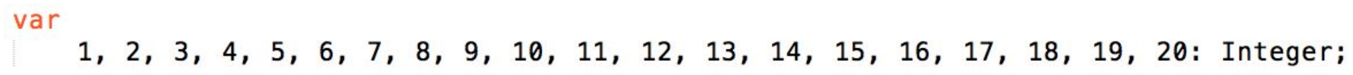
```pascal
function ReadInteger(prompt: String): Integer;
var
    userInput: String;
begin
    userInput := ReadString(prompt);

    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.



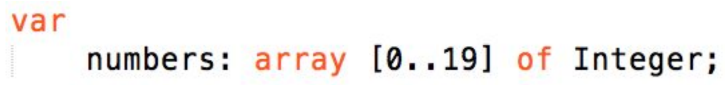
```pascal
function ReadIntegerRange(prompt: String; min, max: Integer): Integer;
var
    userInput: String;
    temp: Integer;
begin
    WriteLn('Enter a number: ');
    ReadLn(userInput);

    while not TryStrToInt(userInput, temp) do
    begin
        WriteLn('Please enter a number: ');
        ReadLn(userInput);
    end;

    result := temp;

    while (result < min) or (result > max) do
    begin
        WriteLn('Please enter a number between ', min, ' and ', max,'...');
        WriteLn('Enter a number: ');
        ReadLn(userInput);

        while not TryStrToInt(userInput, temp) do
        begin
            WriteLn('Please enter a number: ');
            ReadLn(userInput);
        end;

        result := temp;
    end;
end;
```

The implementation of arrays is extremely useful in terms of eliminating excessive variable creation and dealing with multiple values. Let’s say I wanted to make 20 different variables, each one would store an integer. In pascal, that would look like this:



```
var
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20: Integer;
```

It’s long and messy and could even lead to difficulty in tracking variables and their values. By using an array, we can simplify the above stated variable creation so much that it looks like:



```
var
    numbers: array [0..19] of Integer;
```

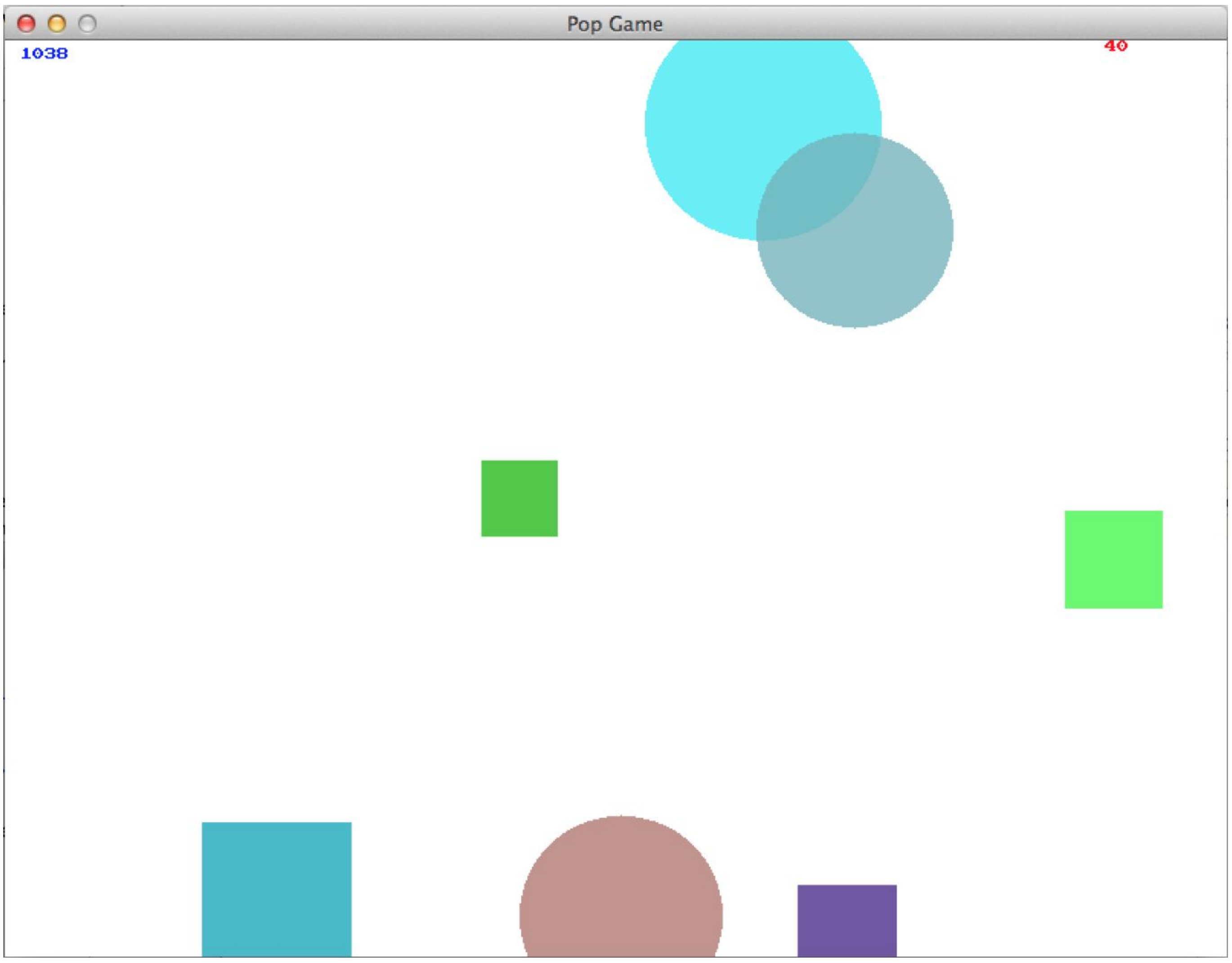
So, now we have an array called numbers, it can take 20 (0 to 19) unique values as integers, similarly to how the 20 declared variables previously could. It’s neat and tidy and the values are easy to track!

I can relate and apply this understanding to the statistics program we made and appreciate that arrays made the overall program and it’s functionality so much easier to implement because the program was dealing with multiple values.

Functional decomposition played an integral part in the creation of PopGame. It was important to understand what the game needed to do and how to implement these needs into smaller building block, which could be used to piece together the bigger picture.

Control was also implemented to have the program perform certain tasks depending on conditions that had to be met. For e.g. control flow is used to make the shapes on the screen disappear only if the left mouse button is clicked within a shape.

I implemented arrays to track the number of shapes which were available for the user to interact with at any given time and custom data types, or records were used to handle the data for each individual shape. For e.g. each shape has a colour, a visible Boolean and a shape type. By using records, all these data types could be condensed into one artefact. It made dealing with data much easier and allowed for ease of programming. Please see the end of this document for the PopGame code.



## 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.

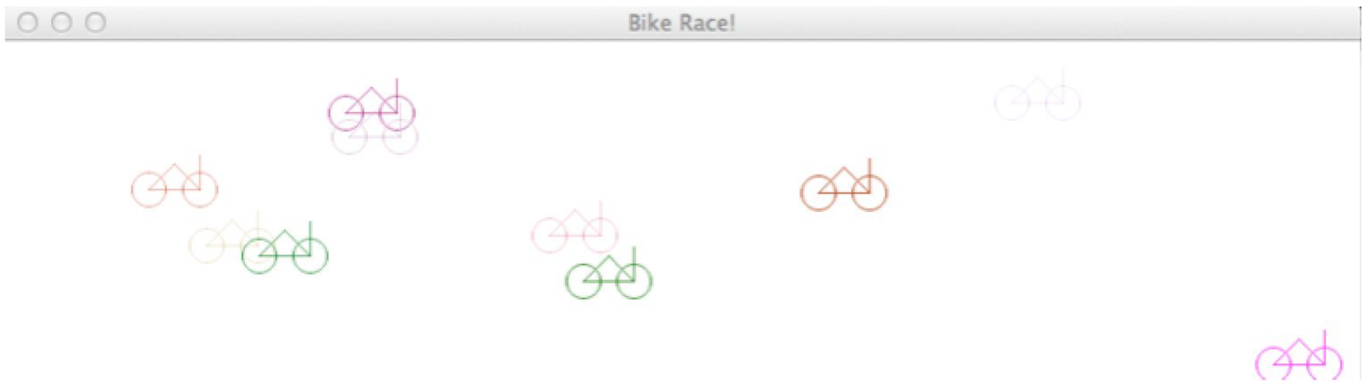
The factors that really contributed to the overall difficulty of understanding were poor indentation practices, ridiculously vague variable names and enough unnecessary repetition to drive anyone insane.

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:

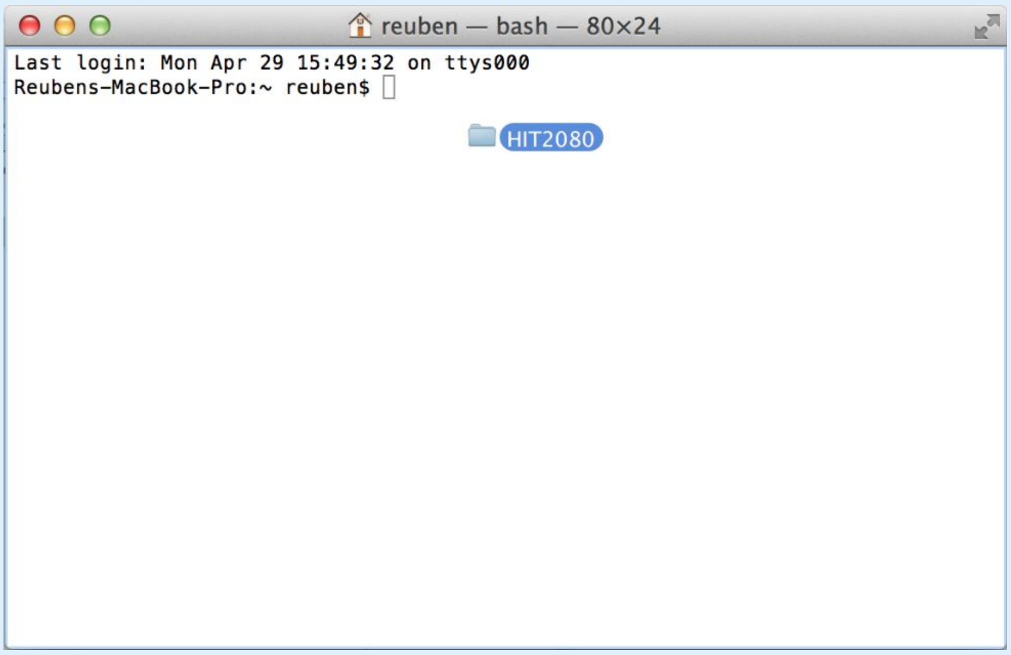
```
if (bikeX > (ScreenWidth() - BIKE_WIDTH)) then
    begin
        bikeX := (ScreenWidth() - BIKE_WIDTH)
    end;
if (bikeY > (ScreenHeight() - BIKE_HEIGHT)) then
    begin
        bikeY := (ScreenHeight() - BIKE_HEIGHT)
    end;
```

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



## Programming Command Line Tools:

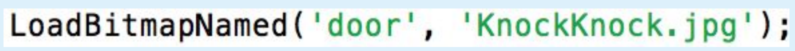
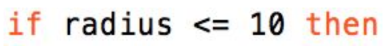
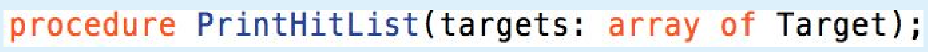
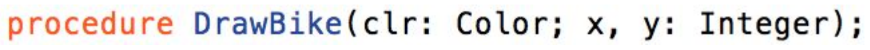
The following tools are necessary to compile and run programs from the command line:

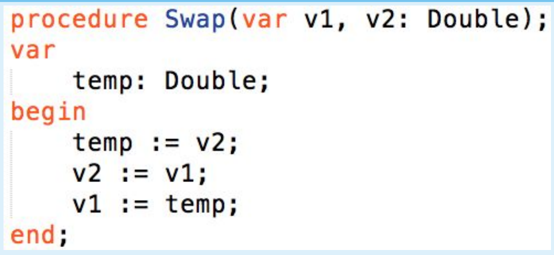
| Command | Description and Example |
|---|---|
| **cd** | Is a terminal command, which stands for '*change directory*'.<br><br>E.g. If my current directory is */Users/reuben/* and I wish to change to the directory */Users/reuben/HIT2080*, this would be achieved by executing the following command in terminal:<br><br>*cd HIT2080/*<br><br>In OSX, you can simply drag your desired directory into terminal as depicted below:<br><br><br>reuben — bash — 80×24<br>`Last login: Mon Apr 29 15:49:32 on ttys000`<br>`Reubens-MacBook-Pro:~ reuben$`<br> |
| **fpc** | Stands for '*Free Pascal Compiler*'. This is the library set, which is used to compile and execute code written in Pascal.<br><br>E.g. I've a file in the directory */Users/reuben/HIT2080* titled *helloworld.pas* that I wish to compile. This is achieved by executing the following command in terminal (assuming my current directory is the above stated):<br><br>*fpc –S2 helloworld.pas* |
| **./helloworld** | This terminal command is used to run a program titled 'helloworld' which has been compiled using the demonstrated understanding from above.<br><br>Assuming I've compiled my Pascal program titled *helloworld.pas* with no errors, I can now run the compiled executable using this command in terminal: |

| | |
|---|---|
| | *./helloworld* |
| **./build.sh** | This command calls upon a bash file titled *build.sh* to execute a predetermined compile script. Let's assume my *build.sh* file is written as follows:<br><br>*#!/bin/sh*<br>*echo "Starting build..."*<br>*fpc -S2 helloworld.pas*<br>*echo "Done!"*<br><br>This particular *build.sh* file will compile a pascal program titled *helloworld.pas* with the *Free Pascal Compiler (fpc).* |
| **./run.sh** | This command calls upon a bash file titled *run.sh* to execute a predetermined execution script. Let's assume my *run.sh* file is written as follows:<br><br>*#!/bin/sh*<br>*echo "Starting execution..."*<br>*./helloworld*<br><br>This particular *run.sh* file will execute a pascal program titled *helloworld* (assuming it has already been compiled and is available for use). |
| **gcc** | Stands for the GNU Compiler Collection. It is a collection of programming libraries, including:<br>• C<br>• C++<br>• Objective-C<br>• Fortran<br>• Java<br>• Ada |

## Programming Terms:

The following terms have specific meanings for software developers. It is important to understand these terms, as they will be used to communicate ideas.

| Term | Description |
|---|---|
| **Statement** | Is an instruction inside a program, which defines an action for the computer to follow when the program is run.<br><br>E.g. This statements action is to call the *SwinGame* procedure *LoadBitmapnNamed( );* with the appropriate parameters being passed.<br><br><br>`LoadBitmapNamed('door', 'KnockKnock.jpg');` |
| **Expression** | Is the term used to identify data, which is used within statements. When a program is run, each expression represents a value, which is used by the statement.<br><br>E.g. This if statement uses the expression *radius <= 10*.<br><br><br>`if radius <= 10 then` |
| **Identifier** | Is used to name or *identify* a section of code. In programming, when new programs, libraries, functions, procedures etc. are created, they are given a unique name, which *identifies them*.<br><br>E.g. This procedure has the identifier *PrintHitList*.<br><br><br>`procedure PrintHitList(targets: array of Target);` |
| **Parameter** | Is a value of a certain data type, which is passed to a function and/or procedure, which requires it for functionality.<br><br>E.g. The procedure *DrawBike* requires three parameters to be passed to it by value in order for it to execute. They are:<br>● *clr* (a colour)<br>● *x* (an integer)<br>● *y* (an integer)<br><br><br>`procedure DrawBike(clr: Color; x, y: Integer);` |
| **Local Variable** | Is a variable (container for a single value), which belongs entirely to the function/procedure in which it is declared.<br><br>E.g. In the procedure *Swap*, the variable *temp* belongs solely to *Swap* and may only be used by *Swap.* |

| | |
|---|---|
| | <br>`procedure Swap(var v1, v2: Double);`<br>`var`<br>`    temp: Double;`<br>`begin`<br>`    temp := v2;`<br>`    v2 := v1;`<br>`    v1 := temp;`<br>`end;` |
| **Global Variable** | Is a variable (container for a single value), which belongs to the entire program, meaning, unlike local variables, they can be accessed/modified from anywhere within the program. The use of global variables is not encouraged because their value can become hard to track. |

## Programming Concepts:

The following concepts are central to procedural programming.

| Concept | Description |
|---|---|
| **Control Flow** | Is used to *control* the order in which code is executed in.<br><br>Control flow statements can be implemented to determine which of two or more branches of code is executed or to create loops.<br><br>E.g. Control flow is implemented by the use of a *case statement* in this example. There are five possible paths to take; each path is completely dependant on unique user input.<br><br>`case option of`<br>`    'e', 'E': Guesses(option, answer, guess, gameCount, 0);`<br>`    'q', 'Q': WriteLn('Bye!');`<br>`    'h', 'H': Guesses(option, answer, guess, gameCount, 0);`<br>`    'm', 'M': WriteLn('Are you retarded?');`<br>`else`<br>`    WriteLn('Shiieeeet. Please try again.');`<br>`end;` |
| **Sequence** | A *sequence* is a specific list of instructions, which can be made up of any number of events; none of the listed instructions can be skipped. Each has to be run in the order in which they are specified.<br><br>`// Call the OpenGraphicsWindow procedure -> effect is a window appears`<br>`//  'Knock Knock...', title to set for the window`<br>`//  800, the width to set the window`<br>`//  600, the height to set the window`<br>`OpenGraphicsWindow('Knock Knock...', 800, 600);`<br><br>`// Call the LoadDefaultColors procedure -> effect is SwinGame colors are initialised`<br>`LoadDefaultColors();`<br><br>`// Call LoadResources -> effect is our images/sounds are loaded into SwinGame for use`<br>`LoadResources();`<br><br>`// Call DrawMySplashScreen -> effect is my custom splash screen is loaded.`<br>`DrawMySplashScreen();`<br><br>`// Call DrawDoor -> effect is the background door is shown... with delay for use to see it.`<br>`DrawDoor();`<br><br>E.g. This particular sequence of code states a specific order in which the procedures are called. The order is as follows:<br><br>1. *OpenGraphicsWindow( );*<br>2. *LoadDefaultColors( );*<br>3. *LoadResources( );*<br>4. *DrawDoor( );*<br><br>The program relies on this sequence to be executed in order to run properly. |
| **Selection** | Is the term which relates to the ability to alter the sequence of code i.e. choosing from two or more branches. |

| | |
|---|---|
| | Remember, a sequence must be followed in the order, which it is written. Implementing selection enables the alteration of this order. |
| **Repetition** | Is the term that refers to a section of code which is repeated a number of times.<br><br>Programmers can implement loops to control repetition. These loops are:<br>● *for* loop – Most often used to loop through each element of an array<br>● *while* loop – Checks the condition for the loop before beginning. This means a *while* loop runs 0 or more times.<br>● *repeat* loop – Checks the condition for the loop after it has run. This means a *repeat* loop runs 1 or more times. |
| **Indentation** | Indentation is an important practice and ensures that, if done correctly, the codes functionality and flow if easy to follow. |

## Structured Programming

**Sequence:**

Referring to the description of *Sequence* from the above table:

"*A sequence* is a specific list of instructions, which can be made up of any number of events; none of the listed instructions can be skipped. Each has to be run in the order in which they are specified."

That being said, sequence is important because it specifies the exact order in which a computer executes the instructions inside a program. If the computer did not follow the order, then the intended result of the program would not be accurate.

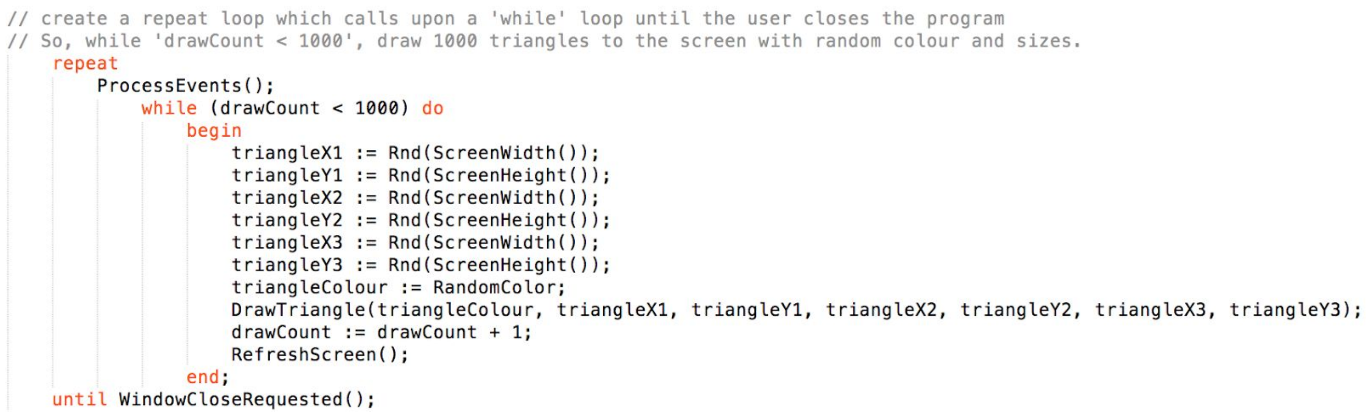
**Selection:**

By using selection, you're not bound by a set sequence of instructions. Selection enables you to select sections of code to be run, providing certain circumstances are met. In relation to what I've covered in Pascal so far, refer to the snippet of code below for an understanding of how I've implemented selection using *if* statements:

```
// If the 'r' key is pressed, assign the 'circleX' & 'circleY' variables random values between screen width and height
// This will draw the circle in a random location on the screen
if KeyTyped(vk_r) then
    circleX := Rnd(ScreenWidth());
if KeyTyped(vk_r) then
    circleY := Rnd(ScreenHeight());
// If the 'c' key is pressed, assign 'clr' a random colour.
// This will change the colour of the circle to a random colour
if KeyTyped(vk_c) then
    clr := RandomColor;
// If the left mouse button is pressed and the mouse pointer is inside the circle,
// assign 'myCircleIsFilled' with the boolean value that it is not.
// E.g. If it is 'True', assign it 'False'. If it is 'False', assign it 'True'
// So, only if the mouse is clicked inside the circle, the circle will become solid, filled with colour
if MouseClicked(LeftButton) and PointInCircle(MouseX(), MouseY(), circleX, circleY, radius) then
    myCircleIsFilled := not myCircleIsFilled;
// If the 'left' key is pressed then subtract '3' from the value of 'circleX'
// This will move the circle on the screen 3 pixels to the left
if KeyDown(vk_Left) then
    circleX -= 3;
// If the 'right' key is pressed then add '3' to the value of 'circleX'
// This will move the circle on the screen 3 pixels to the right
if KeyDown(vk_Right) then
    circleX += 3;
```

**Repetition:**

By using repetition, you can repeat any section of code as many or as little times as you like depending on any dependencies that may or may not have to be met. For example, an excellent scenario of where repetition would be useful is in a small computer game, you want the code that makes the game functional to be repeated for the duration of the game, or until the user closes the application. In regards to how I've implemented repetition in Pascal, refer to the snippet of code below, which uses a *repeat* statement to execute a block of code until the user closes the application:



```
// create a repeat loop which calls upon a 'while' loop until the user closes the program
// So, while 'drawCount < 1000', draw 1000 triangles to the screen with random colour and sizes.
    repeat
        ProcessEvents();
            while (drawCount < 1000) do
                begin
                    triangleX1 := Rnd(ScreenWidth());
                    triangleY1 := Rnd(ScreenHeight());
                    triangleX2 := Rnd(ScreenWidth());
                    triangleY2 := Rnd(ScreenHeight());
                    triangleX3 := Rnd(ScreenWidth());
                    triangleY3 := Rnd(ScreenHeight());
                    triangleColour := RandomColor;
                    DrawTriangle(triangleColour, triangleX1, triangleY1, triangleX2, triangleY2, triangleX3, triangleY3);
                    drawCount := drawCount + 1;
                    RefreshScreen();
                end;
    until WindowCloseRequested();
```

## 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.

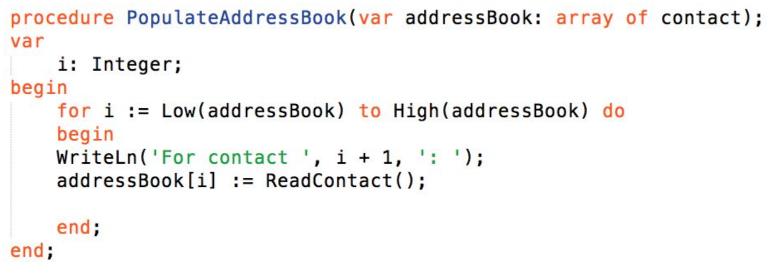
By breaking down large problems into smaller tasks, we can begin to have a better understanding of what it is our program needs to do and how we, as programmers, can implement those needs stress free. These smaller tasks can be the functions and procedures that the program needs to run.

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!

## Iteration

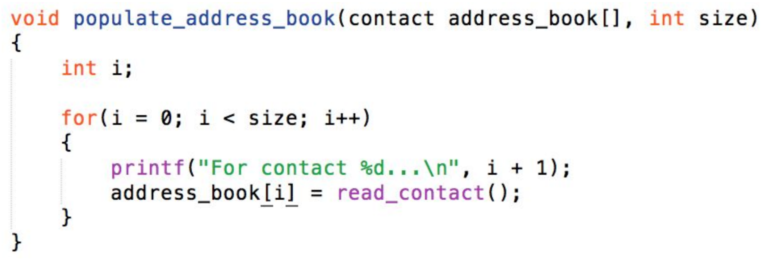
For loops are used in conjunction with a loop counter, which determines the amount of times the loop will be executed. For loops are often used to loop through each element of an array, where the length of the array is the value of the loop counter. Refer to the code examples below:

**Pascal Code**:



```
procedure PopulateAddressBook(var addressBook: array of contact);
var
    i: Integer;
begin
    for i := Low(addressBook) to High(addressBook) do
    begin
    WriteLn('For contact ', i + 1, ': ');
    addressBook[i] := ReadContact();

    end;
end;
```

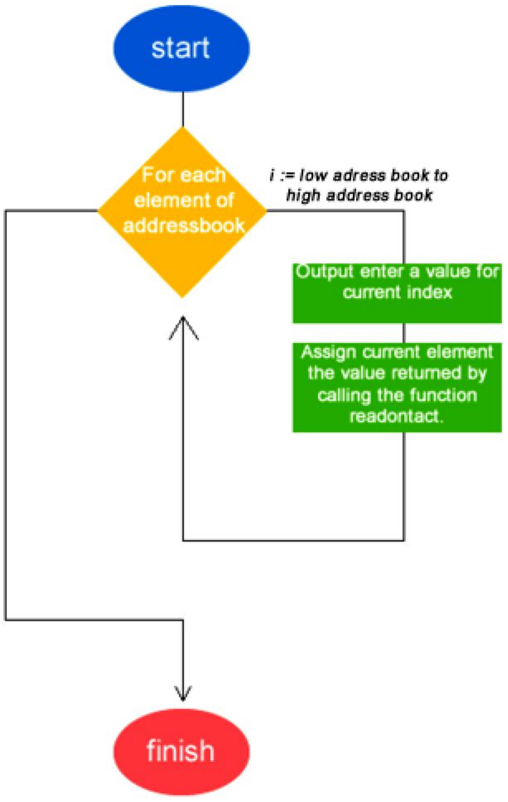
**C Code**:



```
void populate_address_book(contact address_book[], int size)
{
    int i;

    for(i = 0; i < size; i++)
    {
        printf("For contact %d...\n", i + 1);
        address_book[i] = read_contact();
    }
}
```

**How it works:**

This example shows that an array of data type *contact* is being passed to the procedure *PopulateAddressBook/populate_address_book*. The for loop is used to iterate through each element in the array. Each element is assigned the value calculated by calling the function *ReadContact/read_contact*. Please refer to the flow chart below:


start
For each element of addressbook
i := low adress book to high address book
Output enter a value for current index
Assign current element the value returned by calling the function readontact.
finish

## Stack
When a computer runs a program, a section of memory is allocated for the execution of that program.

The allocated memory is split into four different sections. The first section is commonly referred to as the *code area*. This is where the instructions for the program being executed are stored.

The second section is called the *stack*. The *stack* is responsible for storing the information in relation to the current instruction being executed by the computer.

When a computer reaches a procedure/function call within execution of a program, a frame is placed on the stack in relation to the procedure/function call in question. The computer then refers to the *code area* to find the instructions, which belong to that procedure/function and execute each of its instructions in sequence. The stack is also responsible for storing the values that belong to any arrays, variables and parameters that are declared within the procedures/functions being utilized during execution.

Once the current frame on the stack is completed, the computer moves on to the next instruction in the program and repeats the process.

The third section of allocated memory is used to track the values assigned to any global variables declared within the program.

## Heap
The heap is the fourth allocation of memory, which is used to store dynamically allocated data. When a program wants to use dynamic memory allocation, the operating system needs to be informed by the program to set aside enough memory for the value(s), which need to be stored. This is referred to as allocating space. When that allocation of memory and the value(s) it is storing is no longer needed, the operating system can be informed to release,

or free the memory. It is important to always free the memory when it is no longer needed due to performance and resource constraints.

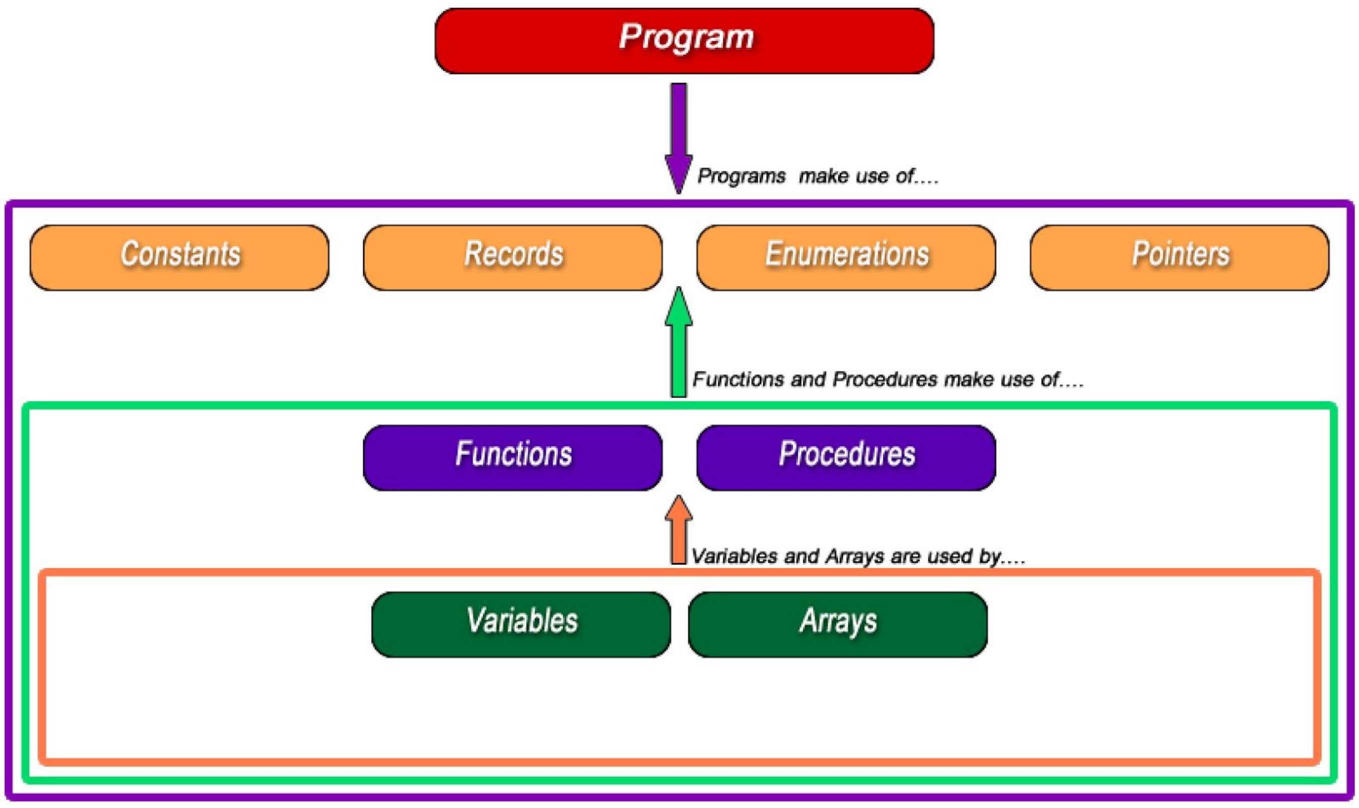
Previously, values that were assigned to variables declared locally were stored in the stack and global variables have their own memory allocated. However, once the stack frame that is assigned to running the current instruction is complete, the variables and the data they contain are lost; they only exist within that certain point on the stack. This is where dynamic memory differs, values can be stored **dynamically** that exist past the point of any sequence of instruction.

Any value, which is stored on the heap can be identified by using a pointer, a special kind of data type which tells the computer that 'the value you are looking for is stored at this location'. When using variables, it is easy to imagine what value they may contain because you can see them coded into your program. Dynamically allocated values however have no coded container as such and therefore can't be visualised because the data they represent is the result of tasks performed when running a program. This is why pointers are important in the use of identifying what data is stored in the heap.

## Programming Artefacts:

Programs can be made from a number of different kinds of artefacts. These include the following that are described below:

- Program
- Procedure
- Constant
- Variable
- Function
- Array
- Record
- Enumeration
- Pointer



## Program

| | **Details / Answer** |
|---|---|
| **A program is …** | A list of instructions, which tells the computer how to execute a task. |
| **A program contains…** | Functions and procedures, which as programmers, we implement to perform certain tasks; these are used in conjunction with parameters (passed in by value or reference) to increase functionality.<br><br>Variables are used to store a single value for use later (to read or to vary); if more than one value needs to be stored, arrays of any size can be implemented.<br><br>Custom data types can be created to cater for specific needs within a program. |

### Example 1: Air Speed Velocity

This example program is designed to calculate the air speed velocity of a bird, based entirely upon the user input.

The user has to specify the frequency at which the birds' wings are beating and the amplitude of each stroke!

You will find the code for this program in both *Pascal* and *C* over the page!

**Pascal Code:**

```pascal
program AirSpeedVelocity;
const
strohaulNumber = 0.33;

function ReadString(prompt: String): String;
begin
        Write(prompt, 'Enter a number for the length of your array: ');
        ReadLn(result);
end;

function ReadDouble(prompt: String): Double;
var
        userInput: String;
begin
        userInput := ReadString(prompt);

        while not TryStrToFloat(userInput, result) do
        begin
                WriteLn('That is not actually a number.');
                userInput := ReadString(prompt);
        end;
end;

function AirSpeed(frequency, amplitude: Double): Double;
begin
        result := Round((frequency * amplitude) / strohaulNumber);
end;

procedure Main();

var
inputAmplitude, inputFrequency, calcAmplitude, airSpeedVelocity: Double;

begin
        WriteLn('Welcome to A Bird''s Air Speed Veolcity calculator');
        inputFrequency := ReadDouble('At what Frequency (hz^2) do your bird''s wings beat? ');
        inputAmplitude := ReadDouble('What is the amplitude of the stroke (cm^3)? ');
        calcAmplitude := (inputAmplitude / 100);
        airSpeedVelocity := AirSpeed(inputFrequency, calcAmplitude);
        Write('Okay, your bird''s airspeed velocity is ', Round(airSpeedVelocity), ' meters per second.');
end;

begin
        Main();
end.
```

**C Code:**

```c
#include <stdio.h>
#define STROHAUL_NUMBER 0.33

typedef struct input_string
{
        char str[250];
} input_string;

input_string read_string(const char* prompt)
{
        input_string result;
        printf("%s", prompt);
        scanf(" %255[^\n]", result.str);
        return result;
}

float read_double(const char* prompt)
{
        input_string line;
        float result;
        char temp;

        line = read_string(prompt);

        while(sscanf(line.str, "%f %c", &result, &temp) !=1)
        {
                printf("Please enter a whole number.\n");
                line = read_string(prompt);
        }
        return result;
}

double air_speed(float frequency, float amplitude)
{
        float result = (frequency * amplitude) / STROHAUL_NUMBER;
        return result;
}

int main()
{
        double input_amplitude, input_frequency, calc_amplitude, air_speed_velocity;

        printf("Welcome to A Bird's Air Speed Velocity calculator.\n");
        input_frequency = read_double("At what Frequency (hz^2) do your bird's wings beat? ");
        input_amplitude = read_double("What is the amplitude of the stroke (cm^3)? ");
        calc_amplitude = (input_amplitude / 100);
        air_speed_velocity = air_speed(input_frequency, input_amplitude);
        printf("Okay, your bird's air speed velocity is %4.2f meters per second.\n", air_speed_velocity);
        return 0;
}
```

## Program Execution:

To gather a comprehensive understanding of what the computer does when a program is run, refer to page 17 of this glossary for an in depth definition of both the stack and heap.

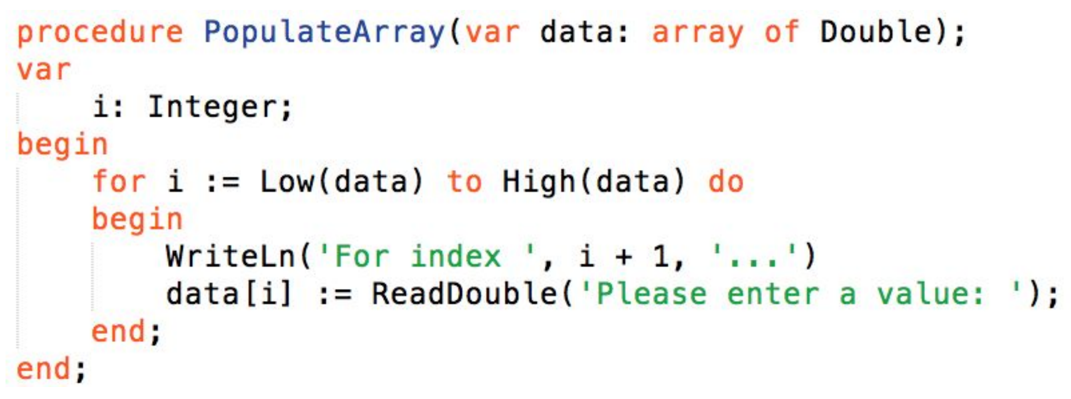
## Procedure

| | **Details / Answer** |
|---|---|
| **A procedure is …** | A set of instructions which, when executed, perform a task.<br><br>Procedures are used to break a program down into blocks; each block having it's own specific task. |
| **Procedures can contain** | Variables, which are used to store a single value calculated within a procedure; arrays can be used to deal with multiple values of the same data type.<br>Repetition can be used to loop a small set of instructions inside of a procedure.<br>Procedures can be declared to take parameters in order to assist with the required functionality. |
| **Procedures are similar to…** | A function, the main difference being that a function returns a calculated value, a procedure does not. |

### Example 1: Populate Array

This example procedure is used to populate an array of data type *Double*. It uses the function *ReadDouble* to calculate the value, which is assigned to each element of the array passed. Note, *ReadDouble* is defined in the *function* section of the glossary below.

**Pascal Code**:



```
procedure PopulateArray(var data: array of Double);
var
    i: Integer;
begin
    for i := Low(data) to High(data) do
    begin
        WriteLn('For index ', i + 1, '...')
        data[i] := ReadDouble('Please enter a value: ');
    end;
end;
```

**C Code**:

```c
void populate_array(double data[], int size)
{
    int i;
    char prompt[17] = "";
    char buffer[3] = "";

    for(i = 0; i < size; i++)
    {
        strncpy(prompt, "Enter value ", 13);
        sprintf(buffer, "%d", (i + 1) % 100);
        strncat(prompt, buffer, 2);
        strncat(prompt, ": ", 2);
        data[i] = read_double(prompt);
    }
}
```

**Procedure Execution:**

When a procedure call is reached during the execution of a program, that procedure is then placed on the stack as a reference point indicating to the computer where the program is at in terms of execution. The instructions for that procedure are then fetched from the *code area*, a section of memory allocated by the computer to store the programs instructions. Once the instructions have been located, they are then executed.

These instructions may include variables and parameters. Refer to the snippet of code below to see how a procedure is being passed parameters:

```c
// Call DrawBike passing it the parameters ColorRed, 50, 10
DrawBike(ColorRed, 50, 10);
```

Now that you can visualise the parameters being passed to the procedure, you can understand from the code below how the procedure (in this case DrawBike) executes in response. The code also outlines how this procedure incorporates variables and their use in execution.

```
procedure DrawBike(clr: Color; x, y: Integer);
var
// Declare variables for the position of the bikes wheels as Integers
wheel1X, wheel1Y, wheel2X, wheel2Y: Integer;
// Declare variables for the positioning of the frame as Integers
frameX1, frameY1, frameX2, frameY2, frameX3, frameY3: Integer;
// Declare variables for the positioning of the handle bars as 2D points on a cartesian plane
barX, barY: Point2D;

begin
    // This section deals with drawing the first wheel (back wheel)
    // Calculate the values for the variables, then assign those vales to their respective variable
    wheel1X := (x + radius);
    wheel1Y := y + (bikeHeight - radius);
    // Draw a circle with the passed colour at the points calculated and assigned to the variables wheel1X and wheel1Y
    // radius defines, you guessed it - the radius for the circle being drawn
    DrawCircle(clr, wheel1X, wheel1Y, radius);

    // This section deals with drawing the second wheel (front wheel)
    // Calculate the values for the variables, then assign those vales to their respective variable
    wheel2X := x + (bikeWidth - radius);
    wheel2Y := wheel1Y;
    // Draw a circle with the passed colour at the points calculated and assigned to the variables wheel2X and wheel2Y
    // radius defines, you guessed it - the radius for the circle being drawn
    DrawCircle(clr, wheel2X, wheel2Y, radius);

    // This section deals with drawing the bike frame
    // Calculate the values for the variables, then assign those vales to their respective variable
    frameX1 := wheel1X;
    frameY1 := wheel1Y;
    frameX2 := x + Round(bikeWidth / 2);
    frameY2 := y + Round(radius / 2);
    frameX3 := wheel2X;
    frameY3 := wheel2Y;
    // Draw a triangle with the passed colour at the points calculated and assigned to the variables
    // frameX1, frameY1, frameX2, frameY2, frameX3, frameY3
    DrawTriangle(clr, frameX1, frameY1, frameX2, frameY2, frameX3, frameY3);

    // This section deals with drawing the bike handle bars
    // Calculate the values for the variables, then assign those vales to their respective variable
    barX := PointAt(wheel2X, y);
    barY := PointAt(wheel2X, wheel2Y);
    // Draw a line with the passed colour from the points calculated and assigned to the variables barX, barY
    DrawLine(clr, barX, barY);
end;
```

## Function

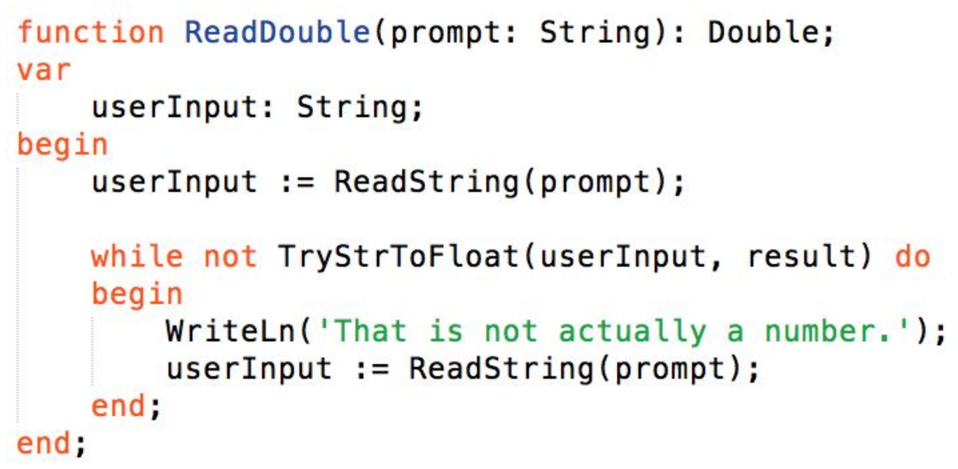
| | **Details / Answer** |
|---|---|
| **A function is …** | A set of instructions which, when executed, return a value. |
| **Functions can contain** | Variables, which are used to store a single value calculated within a function; arrays can be used to deal with multiple values of the same data type.<br>Repetition can be used to loop a small set of instructions inside of a function.<br>Functions can be declared to take parameters in order to assist with the required output. |
| **Functions are similar to…** | Procedures, in the sense that they are both comprised of a small set of instructions, which perform a specific task. |
| **Only difference is…** | Functions return a value! |

### Example 1: Read Double

This example function is used to collect user input and determine whether or not the input matches a value for the data type *double*.

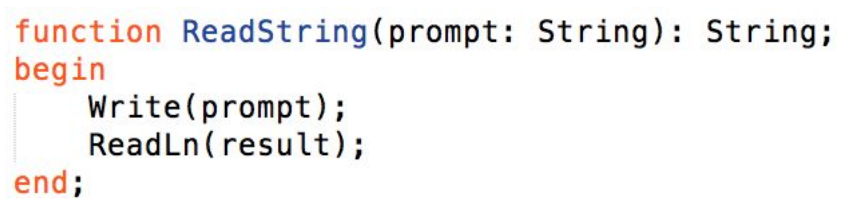
This function uses another function called *ReadString* (provided underneath both examples) to collect the user input, if the input does not match the *double* data type, the user is informed and subsequently must try again!

**Pascal Code**:



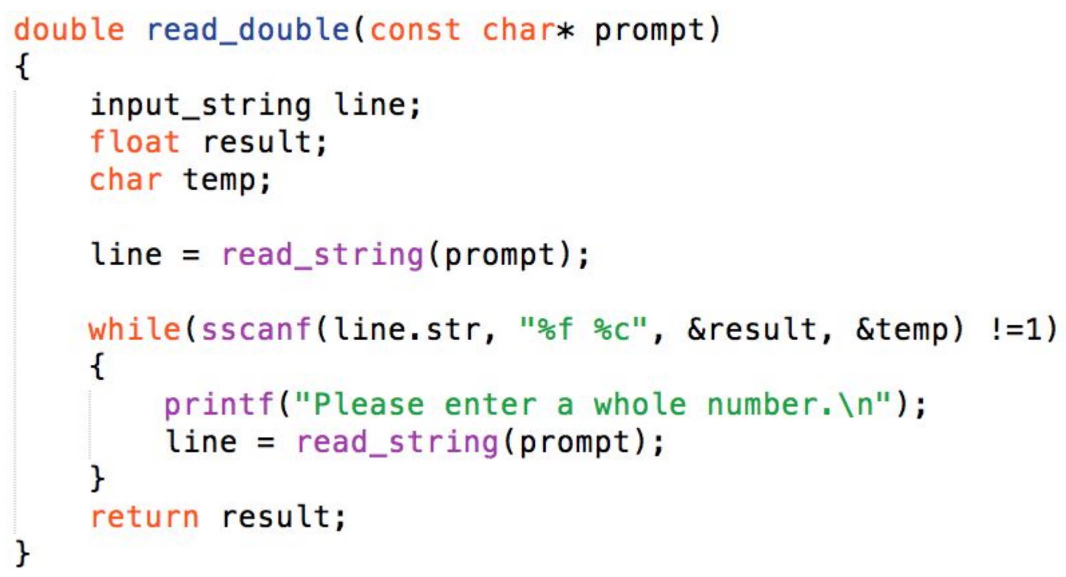
```
function ReadDouble(prompt: String): Double;
var
    userInput: String;
begin
    userInput := ReadString(prompt);

    while not TryStrToFloat(userInput, result) do
    begin
        WriteLn('That is not actually a number.');
        userInput := ReadString(prompt);
    end;
end;
```

Pascal code for *ReadString*



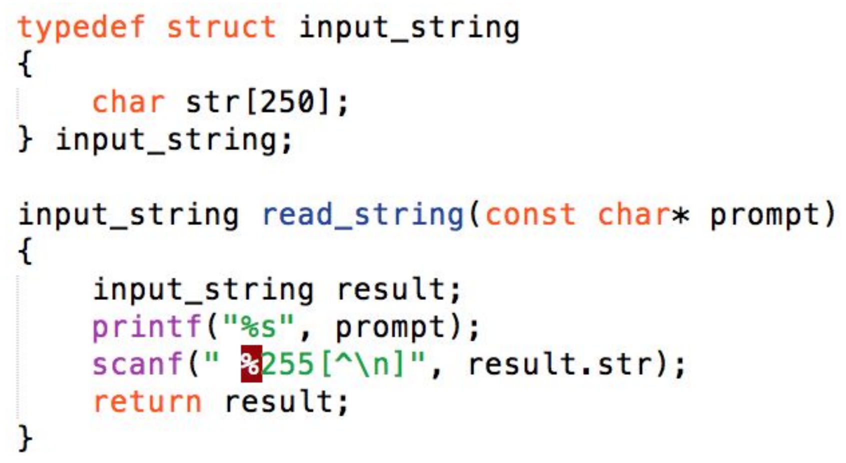
```
function ReadString(prompt: String): String;
begin
    Write(prompt);
    ReadLn(result);
end;
```

**C Code**:



```c
double read_double(const char* prompt)
{
    input_string line;
    float result;
    char temp;

    line = read_string(prompt);

    while(sscanf(line.str, "%f %c", &result, &temp) !=1)
    {
        printf("Please enter a whole number.\n");
        line = read_string(prompt);
    }
    return result;
}
```

C code for *ReadString*



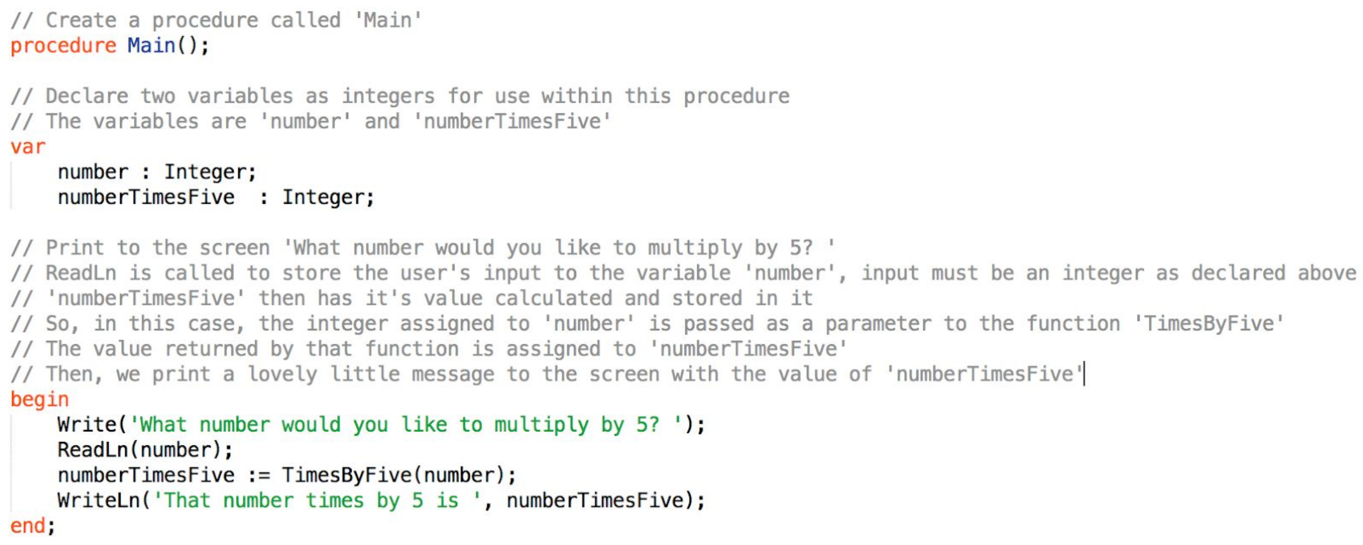
```c
typedef struct input_string
{
    char str[250];
} input_string;

input_string read_string(const char* prompt)
{
    input_string result;
    printf("%s", prompt);
    scanf(" %255[^\n]", result.str);
    return result;
}
```

Note *struct* declaration for *input_string* is included for demonstrational purposes.

## Function Execution:

Functions are executed in a similar fashion as what was described for a procedure execution on page 24. When the computer reaches a function call, that function is then placed on the stack, the instructions for its operation are then located and executed by the computer. Keep in mind, functions return a value where as procedures do not, so that being said, once a function has been executed, a value is going to be returned.

Refer to the snippet of code below, which shows the function NumberTimes5 being called.



```
// Create a procedure called 'Main'
procedure Main();

// Declare two variables as integers for use within this procedure
// The variables are 'number' and 'numberTimesFive'
var
    number : Integer;
    numberTimesFive  : Integer;

// Print to the screen 'What number would you like to multiply by 5? '
// ReadLn is called to store the user's input to the variable 'number', input must be an integer as declared above
// 'numberTimesFive' then has it's value calculated and stored in it
// So, in this case, the integer assigned to 'number' is passed as a parameter to the function 'TimesByFive'
// The value returned by that function is assigned to 'numberTimesFive'
// Then, we print a lovely little message to the screen with the value of 'numberTimesFive'
begin
    Write('What number would you like to multiply by 5? ');
    ReadLn(number);
    numberTimesFive := TimesByFive(number);
    WriteLn('That number times by 5 is ', numberTimesFive);
end;
```

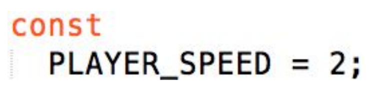
## Constant

| | Details / Answer |
|---|---|
| **A constant is …** | A value within a program, which may never be changed, or remains constant. |
| **Constants are similar to…** | To variables. |
| **The main difference is…** | The value within a variable can be changed, a constant remains constant and the value cannot be changed. |

### Example 1: Player Speed

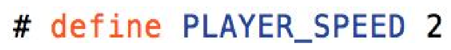
This example constant is used to set the players speed in my custom program project.

**Pascal Code**:



```
const
  PLAYER_SPEED = 2;
```

**C Code**:



```
# define PLAYER_SPEED 2
```

## Variable

| | **Details / Answer** |
|---|---|
| **A variable is …** | A storage location for a single value. |
| **Variables in a program are called:** | Global variables. |
| **Variables in a procedure or function are called:** | Local variables. |
| **Variables passed to procedures and functions are called:** | Parameters. |

### Example 1: Height

This example variable is a demonstration of declaring a variable to store a single value of data type double.

It could be used to store the height of an individual.

**Pascal Code**:

```
var
    height: Double;
```

**C Code**:

```
double height;
```

### Local Variable Creation:

Local variables belong to the function/procedure in which they are declared. When a function/procedure call is reached, the variables that belong to it (if any) are allocated storage space for a single value of their specified data type.

The snippet of code below demonstrates the declaration of variables; these variables will accept a single value as an integer.

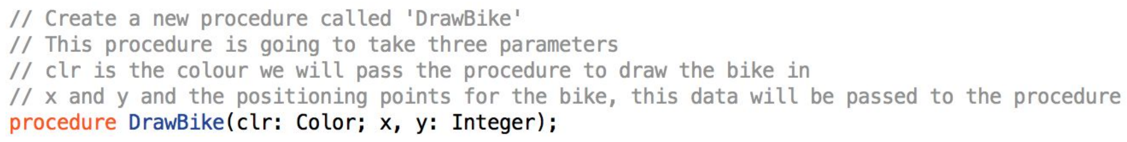
```
// Declare variables for the position of the bikes wheels as Integers
wheel1X, wheel1Y, wheel2X, wheel2Y: Integer;
```

## Parameter Creation:

When a parameter is created, the computer allocates enough memory to store a single value of the required data type.

The snippet of code below demonstrates the declaration of parameters within a procedure; they are:

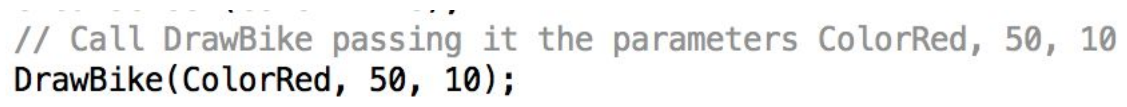
| | |
|---|---|
| clr | – This parameter will take a colour |
| x & y | - These parameters will take integer values only |



```
// Create a new procedure called 'DrawBike'
// This procedure is going to take three parameters
// clr is the colour we will pass the procedure to draw the bike in
// x and y and the positioning points for the bike, this data will be passed to the procedure
procedure DrawBike(clr: Color; x, y: Integer);
```

So, if you refer to the small section below, you can see the procedure DrawBike being passed parameters, which correlate with their data types. The computer:

1. Calculates the value ColorRed and assigns it to the parameter clr
2. Calculates the values 50, 10 and assigns them to them parameters x and y!



```
// Call DrawBike passing it the parameters ColorRed, 50, 10
DrawBike(ColorRed, 50, 10);
```

## Array

| | **Details / Answer** |
|---|---|
| **An array is …** | A storage location for multiple values, unlike variables, which can only store a single value. |
| **When I picture an array…** | I imagine one large container with cells inside of it, each cell capable of storing a single value. |

### Example 1: Data

This example array is used to store multiple values of the data type double. In both cases, the length of the array data is *ten*. Refer to the assignment of the constant value DATA_SIZE in both cases.

**Pascal Code**:

```
data: array [0..DATA_SIZE - 1] of Double;
```

```
const
    DATA_SIZE = 10;
```

**Assigning the constant DATA_SIZE*

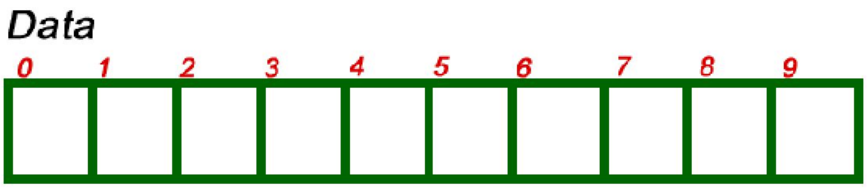
**C Code**:

```
double data[DATA_SIZE];
```

```
# define DATA_SIZE 10
```

**Assigning the constant DATA_SIZE*

**Illustration:**



### Array Creation:

When an array is created, the computer allocates enough storage space so that each element in the array can store a single value of its specified data type. Arrays are a special variable, which can store multiple values!

## Record / Structure

| | Details / Answer |
|---|---|
| **A record/structure is …** | Is a custom data type, which can be implemented to store data, which is outside of scope of any data types native to the programming language in which they are declared. |
| **A record contains …** | Variables and/or arrays of different data types. Enumerations can be used within records also. Each data type declaration within a record is referred to as a field. |

### Example 1: Target

This example record / structure was implemented in the Bounty Hunter program which was made during the semester.

The purpose is to store a name as a string, a bounty value as an integer and a difficulty as a double. These three individual values are combined to create a target.

**Pascal Code**:

```
Target = record
    name: String;
    bounty: Integer;
    difficulty: Double;
```

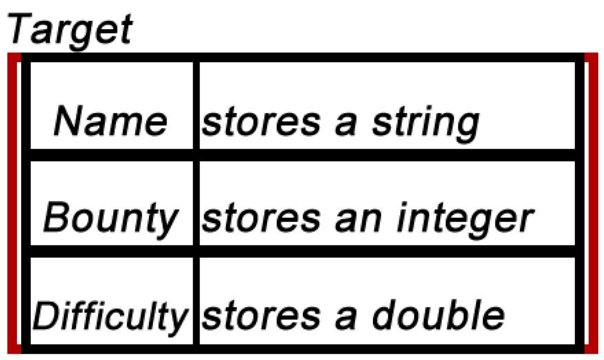
**C Code**:

```
typedef struct target
{
    input_string name;
    int bounty;
    double difficulty;
} target;
```

**Note input_string is a structure. Declaration provided below*

```
typedef struct input_string
{
    char str[250];
} input_string;
```

**Illustration:**



Target

| | |
|---|---|
| Name | stores a string |
| Bounty | stores an integer |
| Difficulty | stores a double |

## Enumeration

| | Details / Answer |
|---|---|
| **A enumeration is …** | Used to offer a number of options for a field within a custom data type. |

### Example 1: Shape Kind

This example enumeration is used within the PopGame program, which was made during the semester.

It is used to offer a selection of three possible shapes, which can be drawn in PopGame. They are circles, rectangles and triangles.

**Pascal Code**:

```
ShapeKind = (CircleKind, RectangleKind, TriangleKind);
```

**C Code**:

```
typedef enum shape_kind
{
    CIRCLE_KIND,
    RECTANGLE_KIND
    TRIANGLE_KIND
} shape_kind;
```

## Pointer

| | **Details / Answer** |
|---|---|
| **A pointer is …** | Is a data type, which refers to a location in memory that has a value stored in it. |
| **When I picture a pointer …** | I imagine that the *pointer* is suggesting, "the value that you are looking for is just around this corner. See! I pointed you in the right direction!" |

### Example 1: Contact Pointer

This example pointer points to a stored value in memory of the contact data type.

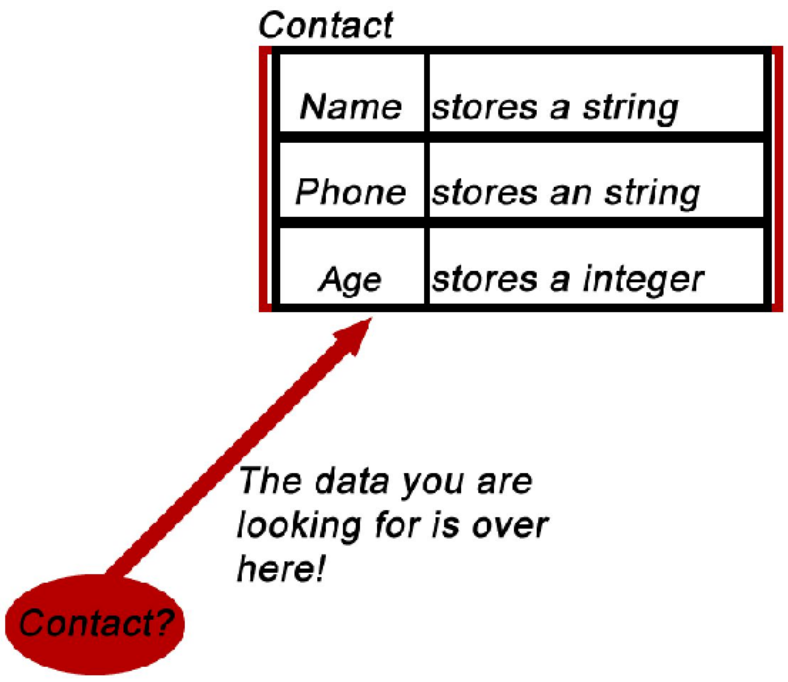
**Pascal Code**:

```
^Contact
```

**C Code**:

```
Contact*
```

**Illustration:**



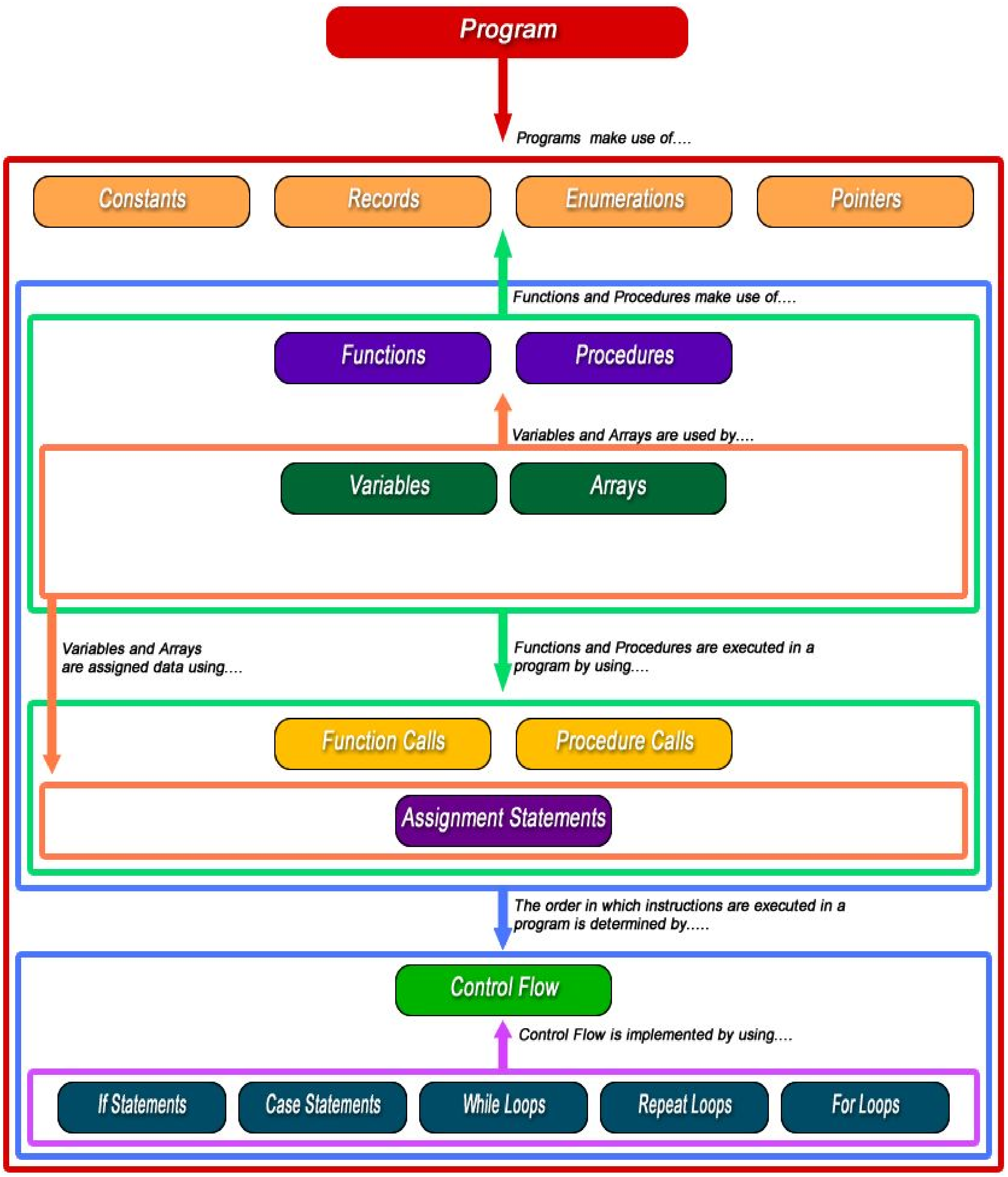
### Pointer Creation and Assignment:

When a pointer pointing to a specific value is created, it stores the location in the computer memory in regards to the value it is pointing to. When this pointer is called upon in a program, it identifies the location of the value being referred to.

## Programming Statements:

Statements are commands that get the computer to perform actions when the code is executed. Each command performs a number of simple steps. The commands listed below are described in the following sections:

- Procedure and Function Call
- Assignment Statement
- If Statement
- Case Statement
- While Loop
- Repeat Loop
- For Loop


Program
Programs make use of....
Constants
Records
Enumerations
Pointers
Functions and Procedures make use of....
Functions
Procedures
Variables and Arrays are used by....
Variables
Arrays
Variables and Arrays are assigned data using....
Functions and Procedures are executed in a program by using....
Function Calls
Procedure Calls
Assignment Statements
The order in which instructions are executed in a program is determined by.....
Control Flow
Control Flow is implemented by using....
If Statements
Case Statements
While Loops
Repeat Loops
For Loops

## Procedure Call

| | Details / Answer |
|---|---|
| **A procedure call …** | Is when a computer has reached an instruction inside of a program to execute a procedure. |
| **Parameters allow you to …** | Pass foreign data to a procedure. Parameters represent values that are calculated and assigned outside of the procedure. |

### Example 1: Draw Game

This example procedure call calls upon the procedure DrawGame/draw_game with the parameter data passed to it.

**Pascal Code**:

```
DrawGame(data);
```

**C Code**:

```
draw_game(data);
```

### Example 2: Player Control Module 1

This example procedure call calls upon the procedure PlayerControlModule1/player_control_module_1 with the parameter player passed to it.

*Note: This procedure is used to calculate the angle between the players sprite and the mouse cursor in my custom program project. Refer to source code for a better understanding.*

**Pascal Code**:

```
PlayerControlModule1(player);
```

**C Code**:

```
player_control_module_1(player);
```

### Actions Performed

When a procedure call is executed the computer performs the following steps:

1. … The computer places the procedure on the stack
2. … The computer then locates that procedures instructions in the code area
3. … The instructions for the procedure are executed step by step
4. … Upon completion, the frame on the stack prior to the procedure call regains control

## Function Call

| | **Details / Answer** |
|---|---|
| **A function call …** | Is when a computer has reached an instruction inside of a program to execute a function. |
| **Is actually an expression as …** | An assignment statement, considering that functions will return a value. |
| **Parameters allow you to …** | Pass foreign data to a procedure. Parameters represent values that are calculated and assigned outside of the function. |
| **The value returned …** | By a function is termed as the *result*. The value of *result* represents the intention of a function's instructions. |

### Example 1: Assignment Statement with Function Call

The value returned from the function call is stored in the variable line using an assignment statement.

**Pascal Code**:

```
line := ReadString(prompt);
```

**C Code**:

```
line = read_string(prompt);
```

### Example 2: Function Call to get parameter value

This example shows a function call calculating a value that is passed to parameter.

**Pascal Code**:

```
CreateShapeAt(RandomScreenPoint(), RandomRGBColor(200));
```

**C Code**:

```
create_shape_at(random_screen_point(), random_rgbcolor(200));
```

### Example 3: Function Call in condition

This example shows two functions being called within the expression of an if statement. They are ShapeAtPoint/shape_at_point(used in PopGame) and a SwinGame function, MousePosition/mouse_position.

**Pascal Code**:

```
if ShapeAtPoint(data.shapes[i], MousePosition()) and MouseClicked(LeftButton) and data.shapes[i].visible then
```

**C Code**:

```
if shape_at_point(data.shapes[i], mouse_poisition()) && mouse_clicked(left_button) and data.shapes[i].visible
```

### Actions Performed

When a function call is executed the computer performs the following steps:

1. … The computer places the function on the stack
2. … The computer then locates that functions instructions in the code area

3. … The instructions for the function are executed step by step and a value is returned

## Assignment Statement

| | **Details / Answer** |
|---|---|
| **An assignment statement …** | Is used to calculate a value and assign, or store it in a variable. |
| **On the left side of the assignment you put a …** | Variable. |
| **On the right side of the assignment you put …** | The value to be calculated. |

### Example 1: Find Mod and Assign it to Result

This example assignment statement calculates the modulus of the value of *size* using the divisor 2. The value is then assigned to result.

**Pascal Code**:

```
result := size mod 2;
```

**C Code**:

```
result = (size % 2);
```

### Example 2: Calculate a Value While Considering BODMAS

This example assignment statement calculates a value whos accuracy depends on mathematical order of operation.

In this example, the value is calculated by performing multiplication first, then addition.

**Pascal Code**:

```
mx := (startMbX + (x * scaleWidth));
```

**C Code**:

```
mx = (start_mbx + (x * scale_width));
```

### Example 3: Calculate a Value Returned by a Function and Assign it to a Variable

This example assignment statement calculates the value returned by the function *ReadDouble(Pascal)/read_double(C)* and assigns it to a variable.

**Pascal Code**:

```
inputFrequency := ReadDouble('At what Frequency (hz^2) do your bird''s wings beat? ');
```

**C Code**:

```
input_frequency = read_double("At what Frequency (hz^2) do your bird's wings beat? ");
```

## Actions Performed

When an assignment statement is executed the computer performs the following steps:

1. … Calculates the value on the right had side of the assignment statement
2. … Assigns the calculated value to the variable on the left hand side

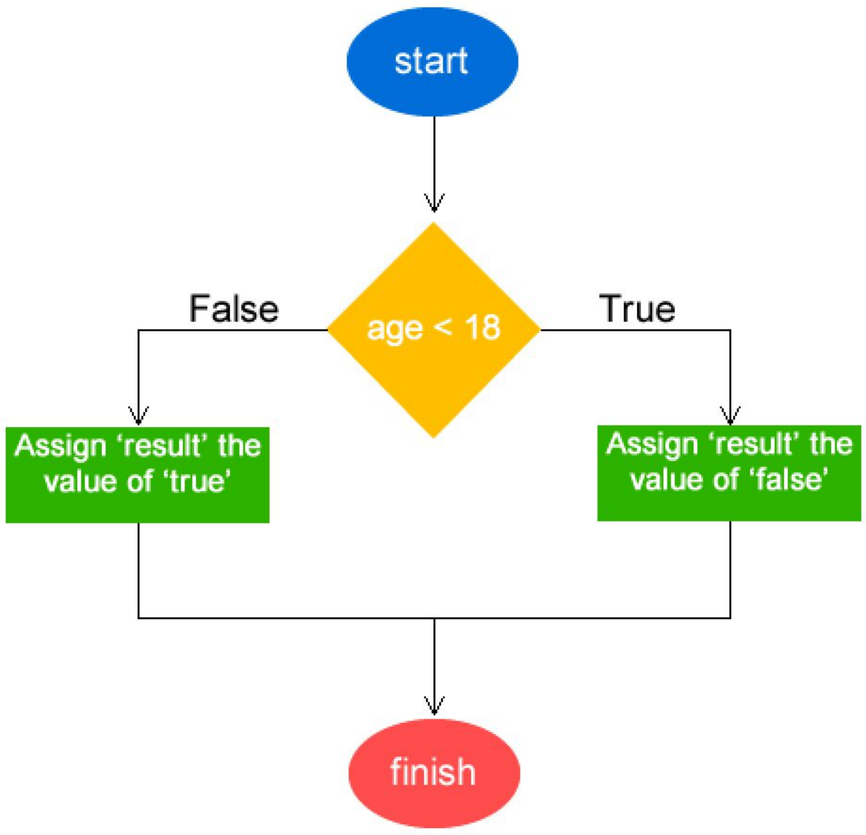
## If Statement

| | Details / Answer |
|---|---|
| **An if statement …** | Is used to execute a selection of code depending on whether the specified expression evaluates to true or false. |
| **This chooses between …** | An if statement usually has two or more branches which can be executed. One branch for when the expression evaluates to true, another for when it evaluates to false. |

### Example 1: Old Enough?

This example if statement is used to evaluate the expression age < 18. If age is less than 18, then the result variable is assigned the Boolean value false, otherwise, result is assigned true.

**Flow Chart:**


start
age < 18
False
True
Assign 'result' the value of 'true'
Assign 'result' the value of 'false'
finish

**Pascal Code**:

```
if (age < 18) then
    result := False
else
    result := True
```

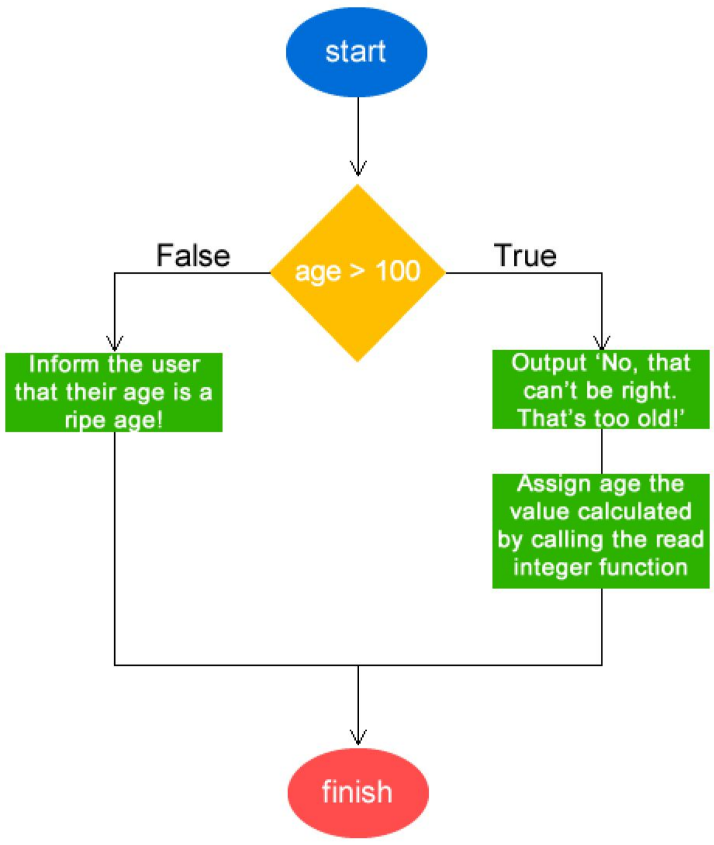
**C Code**:

```
if (age < 18)
    result = false;
else
    result = true;
```

### Example 2: Too Old, Man!

This example if statement is used to determine whether the user is being unrealistic about their age. If age is greater than 100, then the user is reminded of how ridiculous that is and is kindly prompted to re-enter their age. Otherwise, the user is reminded that they're at a ripe age!

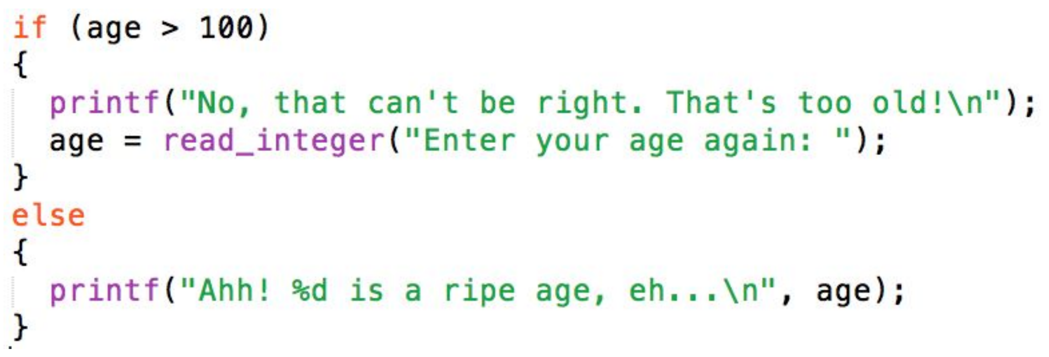
**Flow Chart:**


start
age > 100
False
True
Inform the user that their age is a ripe age!
Output 'No, that can't be right. That's too old!'
Assign age the value calculated by calling the read integer function
finish

**Pascal Code**:

```
if (age > 100) then
begin
    WriteLn('No, that can''t be right. That''s too old!');
    age := ReadInteger('Enter your age again: ');
end
else
begin
    WriteLn('Ahh! ', age, ' is a ripe age, eh...');
end;
```

**C Code**:



```
if (age > 100)
{
  printf("No, that can't be right. That's too old!\n");
  age = read_integer("Enter your age again: ");
}
else
{
  printf("Ahh! %d is a ripe age, eh...\n", age);
}
```

### Actions Performed

When an if statement is executed the computer performs the following steps:

1. … The computer reaches the if statement
2. … The expression is evaluated
3. … If the expression evaluates to true, the statements instructions are executed
4. … If the expression evaluates to false, the statements instructions are ignored

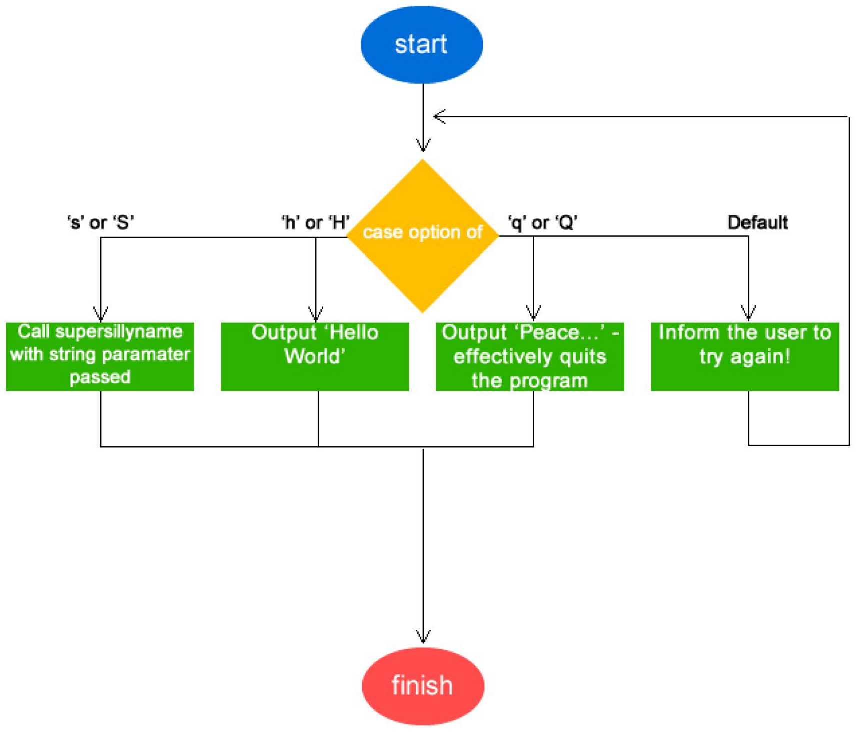
## Case Statement

| | Details / Answer |
|---|---|
| **A case statement …** | Is used to implement control flow in the case of there being multiple options to branch off on. Similar to if statements, case statements require an expression to be evaluated, the evaluated expression then determines the branch to execute. |
| **This chooses between …** | One of the branches possible to follow within the statement. |

### Example 1: A Simple Menu

This example case statement is a demonstration of a simple menu, which could be implemented in any *Pascal* or *C* program.

**Flow Chart:**


start
case option of
's' or 'S'
'h' or 'H'
'q' or 'Q'
Default
Call supersillyname with string paramater passed
Output 'Hello World'
Output 'Peace...' - effectively quits the program
Inform the user to try again!
finish

**Pascal Code**:

```
case option of
    's', 'S': SuperSillyName(name);
    'h', 'H': WriteLn('Hello World!');
    'q', 'Q': WriteLn('Peace...');
else
    WriteLn('Woops ',name, '. Please try again.');
end;
```

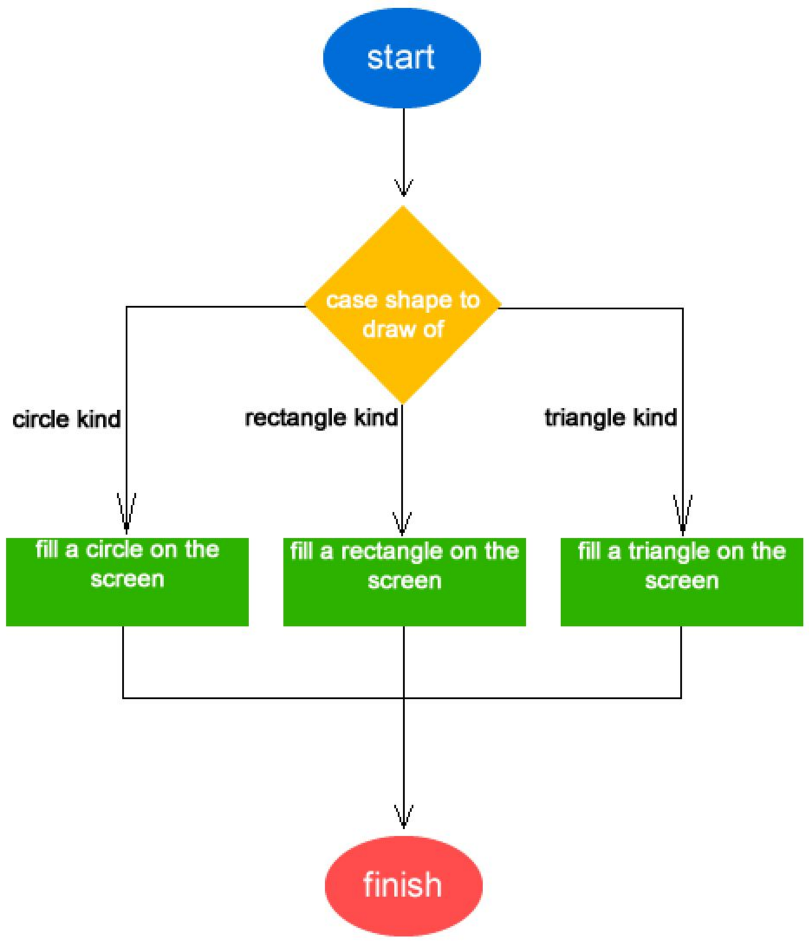
**C Code**:

```
switch(option)
{
    case 's', 'S': super_silly_name(name.str);
    case 'h', 'H': print("Hello World!");
    case 'q', 'Q': printf("Peace...");
    default: printf("Woops %s. Please try again.", name.str);

}
```

### Example 2: Draw a Shape!

This example case statement has three possible branches with no default branch. Each case fills a shape, either a circle, rectangle or triangle.

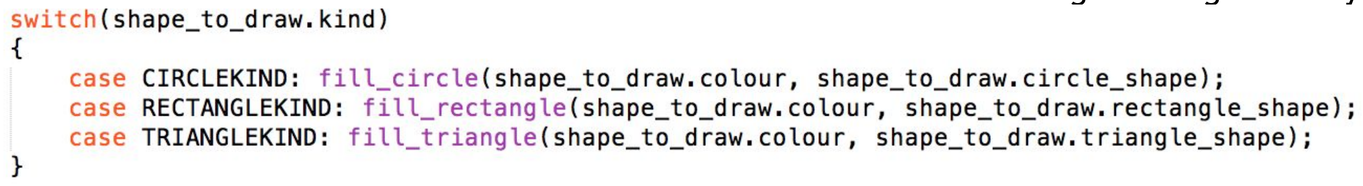
**Flow Chart:**


start
case shape to draw of
circle kind
rectangle kind
triangle kind
fill a circle on the screen
fill a rectangle on the screen
fill a triangle on the screen
finish

**Pascal Code**:

```
case shapeToDraw.kind of
CircleKind:
    FillCircle(shapeToDraw.colour, shapeToDraw.circleShape);
RectangleKind:
    FillRectangle(shapeToDraw.colour, shapeToDraw.rectangleShape);
TriangleKind:
    FillTriangle(shapeToDraw.colour, shapeToDraw.triangleShape);
end;
```

**C Code**:



```
switch(shape_to_draw.kind)
{
    case CIRCLEKIND: fill_circle(shape_to_draw.colour, shape_to_draw.circle_shape);
    case RECTANGLEKIND: fill_rectangle(shape_to_draw.colour, shape_to_draw.rectangle_shape);
    case TRIANGLEKIND: fill_triangle(shape_to_draw.colour, shape_to_draw.triangle_shape);
}
```

## Actions Performed

When a case statement is executed the computer performs the following steps:

1. … The computer evaluates the case statements expression
2. … Depending on the result of the evaluation, a certain sequence of instructions is run
3. … The case statement finishes

## While Loop

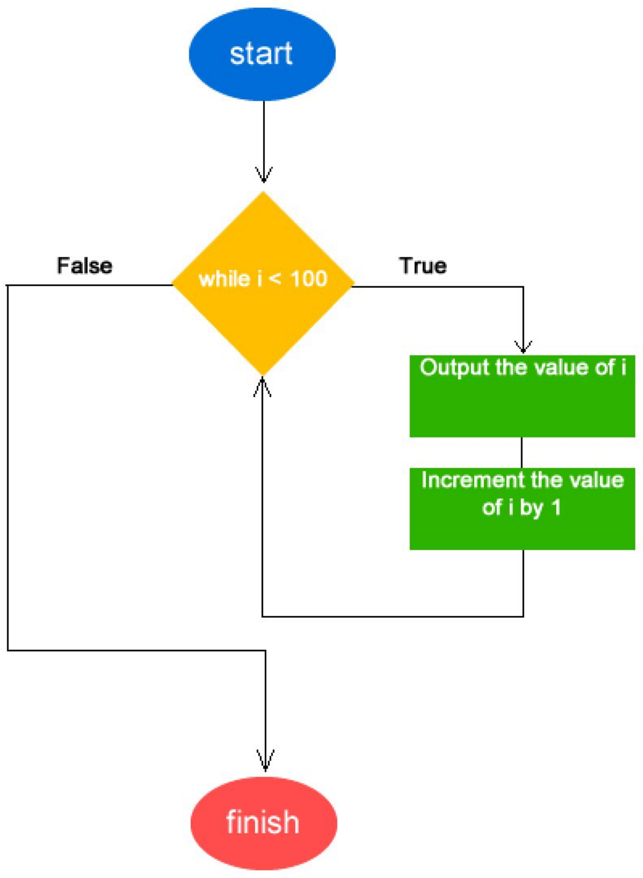
| | Details / Answer |
|---|---|
| **A while loop …** | Is a pre-test loop. The expression for the loop is evaluated before it is executed. |
| **This loop runs its body …** | Zero or more times. |

### Example 1: What Value?

This example while loop will only execute if the value for i is less than 100.

If i is less than 100, the loop will output the value for i and increment i by 1.

**Flow Chart:**



**Pascal Code**:

```
while (i < 100) do
begin
    WriteLn('Value for i is: ', i, '!');
    i += 1;
end;
```

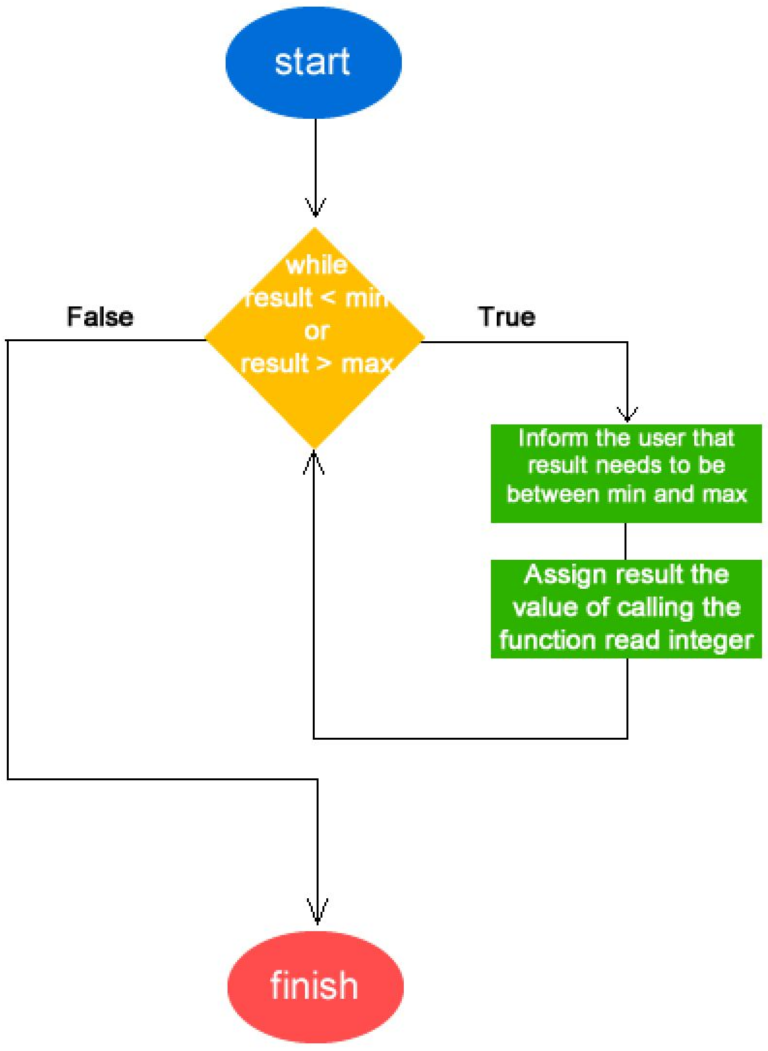
**C Code**:

```
while (i < 100)
{
    printf("Value for i is %d", i);
    i += 1;
}
```

## Example 2: Less than Min or Greater than Max

This example while loop will execute while the value of result is less than max or the value of result is greater than max.

**Flow Chart:**


start
while result < min or result > max
False
True
Inform the user that result needs to be between min and max
Assign result the value of calling the function read integer
finish

**Pascal Code**:

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

**C Code**:

```
while (result < min || result > max)
{
  printf("Please enter a value between %d and %d.\n", min, max);
  result = read_integer(prompt);
}
```

## Actions Performed

When a while loop is executed the computer performs the following steps:

1. … The computer reaches the loop
2. … The loops expression is evaluated
3. … If the expression evaluates to true, the loops instructions are executed and the expression is re-evaluated
4. … If the expression evaluates to false, the loop is not executed
5.

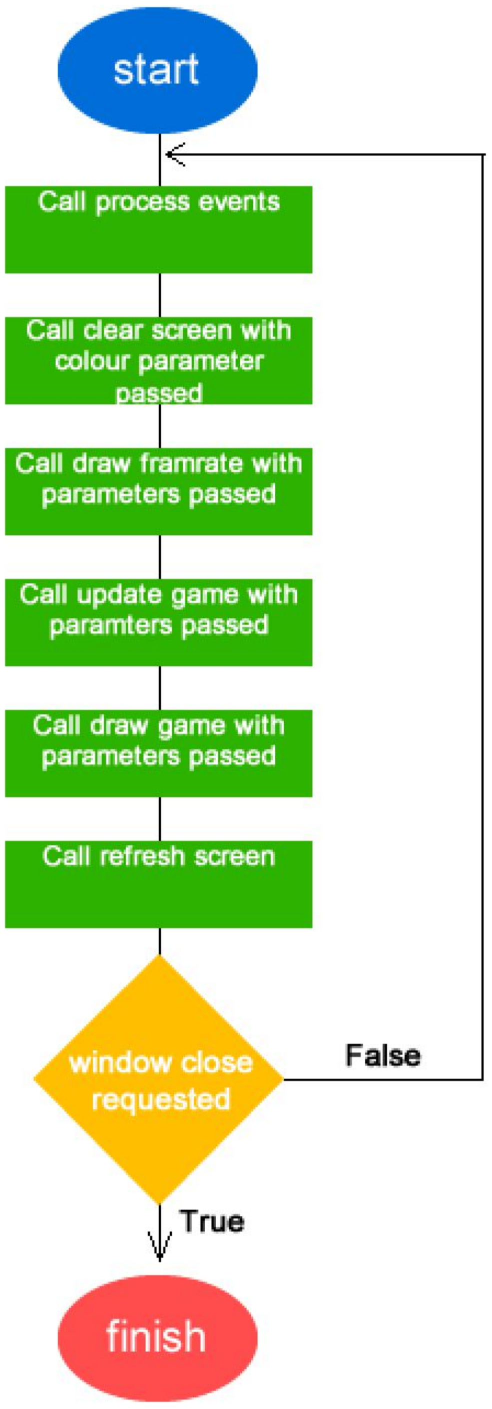
## Repeat Loop

| | **Details / Answer** |
|---|---|
| **A repeat loop…** | Is a post-test loop. The expression for the loop is evaluated after it is executed. In *C*, a do while loop is the same as a repeat loop in pascal. |
| **This loop runs its body…** | At least once. |

### Example 1: The SwinGame Loop

This example repeat loop is used while a SwinGame development is running. It is responsible for executing the code required for the game to function.

**Flow Chart:**


start
Call process events
Call clear screen with colour parameter passed
Call draw framrate with parameters passed
Call update game with paramters passed
Call draw game with parameters passed
Call refresh screen
window close requested
False
True
finish

**Pascal Code**:

```
repeat
  ProcessEvents();

  ClearScreen(ColorWhite);
  DrawFramerate(0,0);

  UpdateGame(player, enemy);
  DrawGame(player, enemy);

  RefreshScreen();
until WindowCloseRequested();
```

**C Code**:

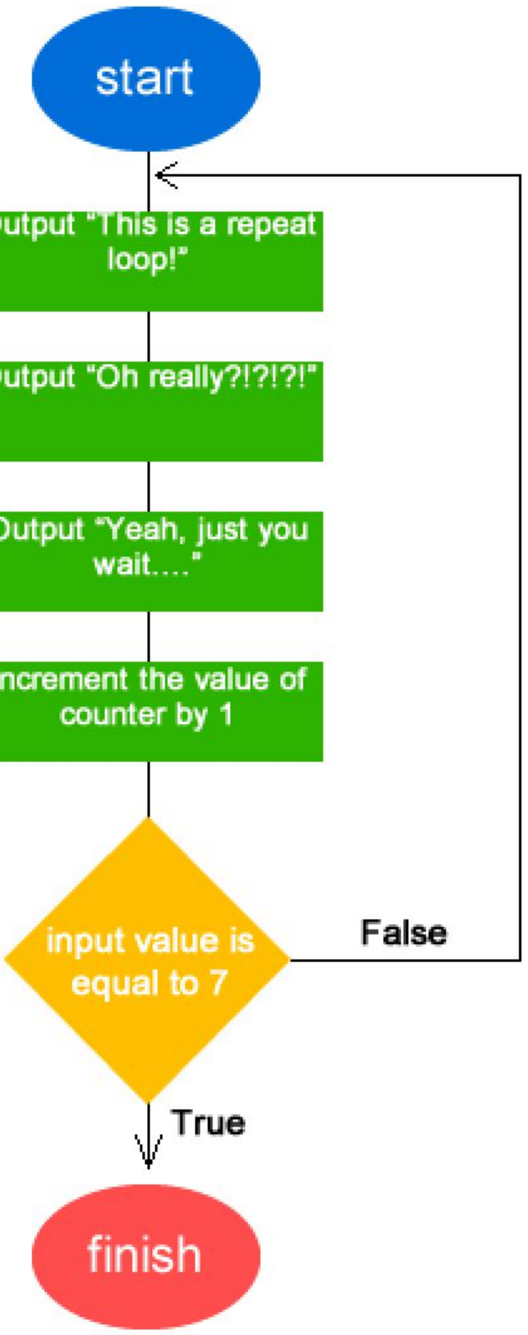
```
do
{
    process_events();

    clear_screen_to(ColorWhite);

    draw_framerate_with_simple_font(0,0);

    update_game(player, enemy);

    draw_game(player, enemy);

    refresh_screen();
} while ( ! window_close_requested() );
```

### Example 2: Tell me that Value!

This example repeat loop points out the obvious, that it is indeed a repeat loop. It will repeat until the value of counter is equal to 7.

Note that this loop is fundamentally flawed, on account of repeat loops are post test loops – if the value is already 7 before the loop is reached, it will still run.

**Flow Chart:**


start
Output "This is a repeat loop!"
Output "Oh really?!?!?!"
Output "Yeah, just you wait…."
Increment the value of counter by 1
input value is equal to 7
False
True
finish

**Pascal Code**:

```
repeat
  WriteLn('This is a repeat loop!');
  WriteLn('Oh really?!?!?!');
  WriteLn('Yeah, just you wait....');
  counter += 1;
until (counter = 7);
```

**C Code**:

```
do
{
    printf("This is a repeat loop!\n");
    printf("Oh really?!?!?!\n");
    printf("Yeah, just you wait....\n");
    counter += 1;

} while ( count != 7 );
```

## Actions Performed

When a repeat loop is executed the computer performs the following steps:

1. … The computer executes the loops sequence of instructions
2. … The computer evaluates the loops expression at the end of the execution
3. … If the expression evaluates to true, repeat the loop
4. … If the expression evaluates to false, finish the loop

## For Loop

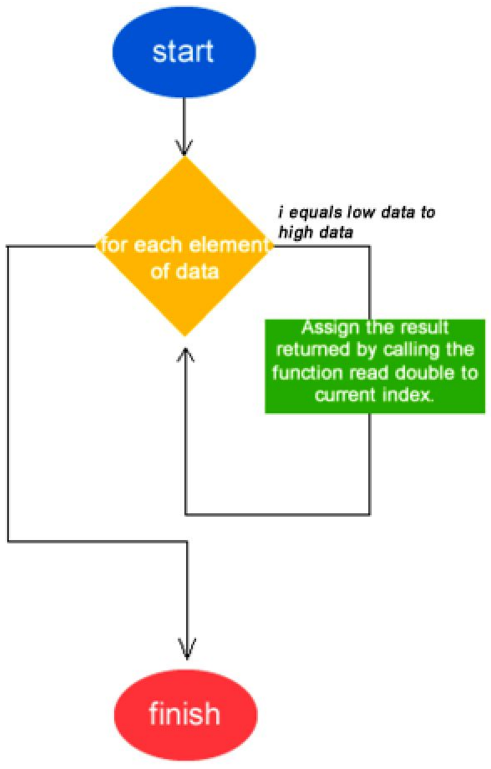
| | **Details / Answer** |
|---|---|
| **A for loop …** | Allows code to be executed over and over again. They are usually used in conjunction with a loop counter, or in other words, a variable, which tracks the number of times the loop runs. |
| **For loops are typically used with…** | With arrays. I stated above that for loops use a kind of loop counter, think of the length of an array; this length value can be used as a counter value for a for loop. |

### Example 1: Populate an Array

This example for loop iterates through each element of an array of data type double and assigns a value at each index using the read double function.

**Note that C can't calculate the length of an array like Pascal. A variable is used to store a value, which represents the length of an array, this variable is then used in iteration. In the C example provided, this variable is called size.*

**Flow Chart:**


start
for each element of data
i equals low data to high data
Assign the result returned by calling the function read double to current index.
finish

**Pascal Code**:

```
for i := Low(data) to High(data) do
begin
    WriteLn('For index ', i + 1, '...')
    data[i] := ReadDouble('Please enter a value: ');
end;
```

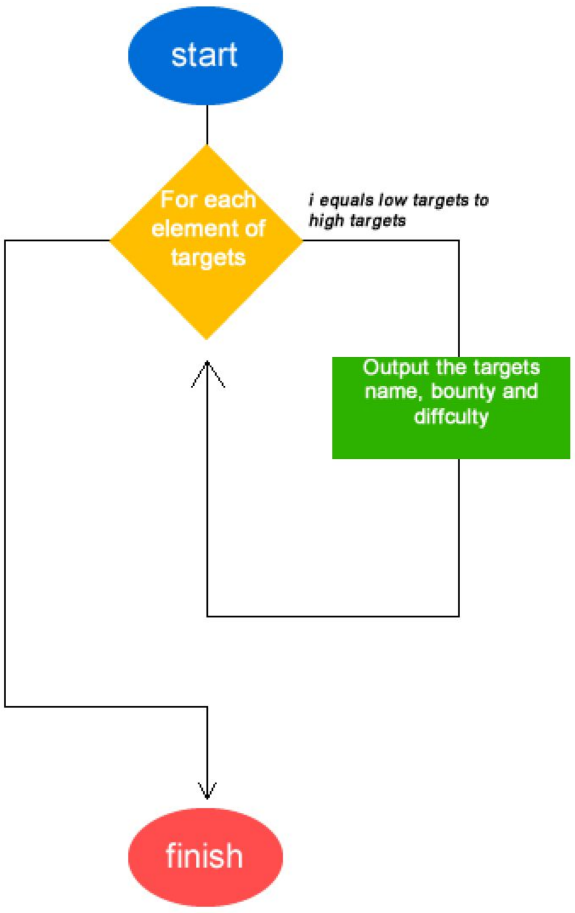
**C Code**:

```
for(i = 0; i < size; i++)
{
    strncpy(prompt, "Enter value ", 13);
    sprintf(buffer, "%d", (i + 1) % 100);
    strncat(prompt, buffer, 2);
    strncat(prompt, ": ", 2);
    data[i] = read_double(prompt);
}
```

## Example 2: Print Target Details

This example for loop is used in the Hit List program to iterate through each element of an array of target and print the targets details!

**Flow Chart:**


start
For each element of targets
i equals low targets to high targets
Output the targets name, bounty and diffculty
finish

**Pascal Code**:

```
for i := Low(targets) to High(targets) do
begin
    Writeln(i + 1, ': Name - ', targets[i].name,
    ', bounty $', targets[i].bounty,
    ', difficulty (', targets[i].difficulty:4:2, ').');
end;
```

**C Code**:

```
for(i = 0; i < size; i++)
{
    printf("%d: %s.\n $%d in bounty. Difficulty rating: %f\n",
    (i + 1),
    targets[i].name.str,
    targets[i].bounty,
    targets[i].difficulty);
}
```

## Actions Performed

When a for loop is executed the computer performs the following steps:

1. … The computer determines how many times the loop is to be executed
2. … For each execution (iteration) the computer executes a sequence of instructions
3. … At the end of each execution the computer evaluates the loop expression
4. … If the expression evaluates to true, execute the loop
5. … If the expression evaluates to false, finish the loop