A BEGINNERS GUIDE TO PROGRAMMING

Book 1

Programming and Coding Overview

Alex Maddern

23 October 2022

Table of Contents

Licence

Written by Alexander Maddern.

With many thanks to Daniel Moore for proof reading and corrections as well as the many ideas, examples and suggestions that have been provided.

This document is provident in the hope that it will be provide a useful overview of the concepts that are covered and that many concepts will only be covered in part or brief. The author does not except liability for the accuracy of the content provided within this document. The reader should seek to obtain documentation for the specific programming languages and platforms used within this document. It is recommended to use a sandboxed virtual environment to test all of the examples that have been provided.

Revisions

Version	Date	Notes
Draft V 0.1	16/12/2021	Copy basic outline
Draft V 0.2	17/12/2021	Re-organise and revise current material.
Draft V 0.3	12/01/2022	Minor edits (Daniel - proof and grammar corrections)
Draft V 0.4	26/01/2022	Include TAFE Unit C example
Draft V 0.4	29/01/2022	Document archived, replaced with MS Office version.
Draft V 0.5	29/01/2022	Included all csv examples. Additional code examples. Minor visual changes.
Draft V 0.6	14/02/2022	Added examples "Decisions". Added section "Unicode".
Draft V 0.7	15/02/2022	Added example "Loops". Tidy example naming. Add source files. "Beginers_Guide_To_Programming_Source.7z"

Draft V 0.8	20/02/2022	All code is cross platform and tested under Windows x64(W10) and Linux x64 (Ubuntu 20.04 LTS). Tidied comment spacing and reduced most code to < 80 character width.
Draft V 0.9		Include additional content and tidy.
Draft V 0.10	27/02/2022	Added example “Functions”. Additional tidying.
Draft V 0.11	28/02/2022	Expand missing explanations. Additional document tidying.
Draft V 0.12	01/03/2022	Expand missing explanations. Additional document tidying.
Draft V 0.13	02/03/2022	Expand missing explanations. Additional document tidying.
Draft V 0.14	06/03/2022	Add explanation for “Types”. Code example.
Draft V 0.15	08/03/2022	Complete additional items explanation. Document tidying.
Draft V 0.16	10/03/2022	Additional contend under “H1 Additional Items”. Document tidying.
Draft V 0.17	29/04/2022	Corrections to Practical Task source. Remove TABs from all source.
Draft V 0.18	01/05/2022	Complete “Additional Components”.
Final V 0.19	01/05/2022	Final Draft – Limited Public Preview.
Final V 0.20	12/05/2022	Added Debug Print examples. Added Dev-C++ and Thonny breakpoint instructions.
Final V 0.21	16/05/2022	Correct Linux xmessage print format in “Functions.bas”. Fix 16bit error check in reterr in C FB Py (highbyte = reterr>>8).
Final V 0.22	16/05/2022	Add Windows and Ubuntu GUI Debug MessageBox examples.
Final V 0.22	19/05/2022	Added examples for error checking and input validation. Corrected the coding in many examples.
Final V 0.23	23/10/2022	Changed example source license to MIT/MIT-0

TODO:

PDF version looses line return formatting of source code during copy/paste. Use provided source files.

Preface

About me

I first started in electronics and hardware based “Logic Programming” in my early childhood.

During my first years in Secondary School (1978-) my math teachers introduced me to some 8-bit Ohio Scientific personal computers. Although pre-written software could be loaded from disk, it was supplied in source code for use in one of the many BASIC language interpreters used on computers of the time.

Personal computers (PC) were not a thing yet, and very few people would have even seen a PC let alone owned one!

My math teacher gave me some instructions of the dos and don’ts, a brief overview of the programming guide, and access to some software that he had collected on disk, so off I went clumsily loading software and attempting to write my own short pieces of code during my lunch breaks. He was an excellent instructor and would drop by and check on me, giving some simple advice and to help me to create neater routines in my code. This really was an excellent experience for me.

I went on to write many small text-based applications including some math calculators, guessing games, like hangman, and later a rewrite of “The Labyrinth”, a text-based adventure game, and later an upgrade of the ASCII graphic “Star War” game.

Working on this simple system I learned a great deal about computer hardware as well as the essential programming techniques used to make use of the hardware. In the next few years I also went on to port a text-based adventure game called “Hunt The Wumpus” from Apple BASIC to MS BASIC. It was during this exercise that I discovered the difference between *Programming* and *Coding*.

From Grade 10-12 I enrolled in my school's elective class “Advanced Computer Mathematics”, which essentially took the higher level math skills over to the computer programming world. Programming was taught as an extension of mathematics. Computers even today are very heavily grounded in mathematical logarithms and numeric manipulation, and what I had learned set me up for the ownership of a long string of PCs. As the PC world grew in size and capability, the possibilities for programming grew and I grew along with it.

Most of my earlier PC programming experience was based around the many different versions of BASIC as well as the advancements in the Disk Operating Systems (DOS) of the time. Although I had never coded in machine languages, such as HEX, Assembly, or C I had already obtained a solid grounding in the principles of computer programming.

Today I have some familiarity with a large portion of the available programming languages, meaning I can read and understand the code routines even if I have never written the code in that language. I am also confident in a number of modern mainstream compiler and scripting languages that I use on a regular basis.

The majority of my experience has been in the many variants of BASIC, as well as the C programming languages, and this will often reflect in my style or how I explain programming concepts.

The larger majority of mainstream programming languages have a strong basis in C and BASIC language constructs and syntax. Learning one of these languages makes it less difficult to read and

learn an alternative language. The similarities of the basic principles of coding structure will quickly become apparent from one language to the next.

On learning computer programming

I believe that learning computer programming should follow a pathway that does not appear in the modern education system; at least it doesn't appear to in Australia. In today's world of computer technology learning to code is analogous to learning to read and write and as much as we would struggle in many ways without basic reading and writing skills this is also true of the world of computer technology. Having at least a minimum understanding of how a computer device works, including the hardware in programming, allows us to make sound decisions on the technology we choose to use in both industry and our everyday lives.

Computer mathematics

Computer mathematics should be the first step in the learning pathway.

This doesn't focus on any programming language in particular, but focuses on the underlying mathematical principles as they apply to computer technologies. This can be facilitated via the use of a simple medium-level language, such as traditional BASIC or even C.

The student would learn the fundamental skills of computer math and the most fundamental of coding principles. At this point a student would be capable of understanding any future programming language based upon those principles, covering most common languages today.

Learn to program

There seems to be a common focus in Australia on learning high abstraction level languages.

Although this may appear to offer a "fast track" to high level outcomes, it can actually cripple the academic future of that student.

It is somewhat analogous to learning the tricks of the trade without learning the trade.

When a problem arises that is outside of the narrow learning paradigm of the high level language, the student will be lost in the problem solving process as they were not equipped with the underlying knowledge to begin with.

C

C is for some reason avoided as a "difficult" language to learn, and I cannot for the life of me understand the reasoning in this.

C at its most basic level, aka C the programming language, is really quite simple, concise, and easy to learn. The learning curve is much the same as learning traditional BASIC, and the C syntax is heavily influenced by traditional BASIC. If you can code in traditional BASIC, you can code in C.

Easy to learn, difficult to master. C and C++ take some learning effort to master. Ironically, this is true of any programming language of any abstraction layer.

The difficulty is not inherent in the language but in the libraries and interaction with the variety of hardware and operating systems (OS). The thing that I find most relevant is that nearly all OSs, such as the kernel and other parts of Microsoft Windows, Linux, MacOS, etc., are written predominantly in C.

Therefore, any OS naturally becomes an extension of the students' understanding of the C language.

Most high level languages interact with the exposed C runtime almost exclusively, so you still need to have an understanding of the C language and C application programming interfaces (API) as they are exposed by the OS.

Observations

This fast tracking and taking "short cuts" to a high level outcome may actually inhibit student progress in my view.

The small learning curve needed to gain an understanding of computer mathematics and C programming fundamentals allows the student to progress more intuitively through the high level of abstraction found in other languages, OSs, APIs, and libraries.

Many nations take this pathway working from a lower level of abstraction, and it seems their trained engineers and computer coders may be outperforming students coming out of the Australian education system.

I was fortunate enough to have gone through the pathway that I mentioned earlier, and in all honesty, I am glad every day that I had the good fortune that this computer mathematics pathway was available to me in the late 70s and 80s.

I am writing this guide in the hope that students, such as yourself, might follow the pathway I did earlier, to gain a great foundational understanding of computer mathematics and programming, setting you up for a possible lifetime of programming success.

Alex Maddern

12/01/2022

Introduction

Programming methods

Before I continue, I must make it clear that this guide and the others in this series look at coding from an *Imperative Procedural Orientation*. What does that mean? This refers to the programming methodology and how the actual code and manipulation of data in the program is structured. There are three major programming paradigms in common use:

- Object-Oriented Programming (OOP)
- Procedural Programming
- Functional Programming

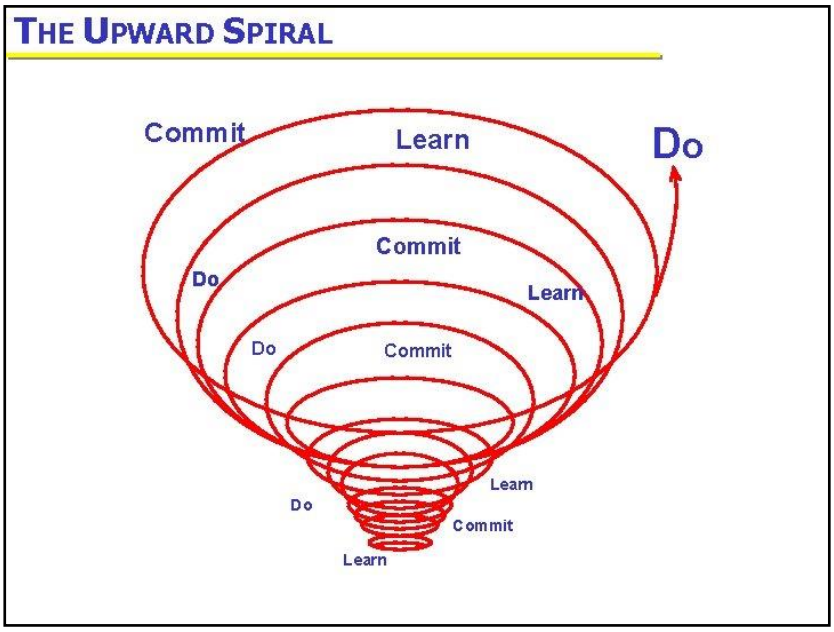
I won't go into detail about the differences here, but you can find out a little more about these programming paradigms on the web using your favourite search engine, or by reading Functional vs. Procedural vs. Object-Oriented Programming.

What you will find is that the three major styles will blend a little in practice, and one style will often inherit parts from another style.

I will be following an *Imperative Procedural* approach, meaning that programming is approached as an *inline* or *step by step* set of instructions based around calls to Routines, Subroutines, and Functions, although some components of Functional programming will also be used.

You will notice that I will appear to have repeated the same topics throughout this booklet. This has been done by design. The learning process in computer programming has a natural tendency to follow an upward spiral of learning. Each time we come back over the same topic we do so at a slightly more knowledgeable level, gradually adding more skills each time.

"We shall not cease from exploration, and the end of all our exploring will be to arrive where we started and know the place for the first time." -- T. S. Eliot



Stephen Covey – 7 Habits

Programming vs Coding

Although the terms programming and coding are often used interchangeably, programming is the overall process of creating an application for the computer hardware, which will often entail more than just the source code, whereas coding is the practice of writing the individual lines of source code.

Programming is generally a language neutral process that describes the problem and solution as well as the logical flow of the programming solution. In practice though, there will always be some blurring of the line between programming and coding as a particular language may be identified as part of the solution.

Programming may include an outline of the problem as well as proposed solutions. The programming solution may incorporate explanations, diagrams, flow charts as well as other methods of explaining the steps toward implementing a solution.

One of the most common representations is the use of a flowchart that will describe in some detail the program flow as well as the logic that needs to be implemented in a coded form.

Programming languages similarities and differences

Four examples of Hello World user input and math functions

Don't try to understand how the following code examples work for now - just look over each and look for similar patterns in the way the code is written.

The first example is a small section of Assembled Machine Code from the C source in the third example.

The second example in Assembly is the exact same application as the C application following it.

The C compiler converts the C source code to Assemble source, and then converts it to Machine Language.

The FreeBASIC and Python source code are the same application written in other languages.

If you look over each, you will see that most of the programming statements reoccur in a similar position in each language. Although there may be some slight differences in the way keywords are spelt in the different code examples, they are all performing a similar task.

As we work through this booklet I will focus on the underlying coding technique rather than any specific language. Hopefully this will help the reader to look at any common language and be able to see the same methods and principles in use.

N.B. The Code highlighting used in the examples is created using the Notepad++ "Export to RTF" plugin. The RTF can be then pasted into a word processor keeping the coloured highlighting.

Machine language (HEX, .0, .exe)
Code: (Sample string section only) "Example1.o"

50 6C 65 61 73 65 20 65 6E 74 65 72 20 79 6F 75 72 20 6E 61 6D 65 3A 20 00 25 73 00 00 00 00 00 48 65 6C 6C 6F 20 25 73 2E 20 43 61 6E 20 79 6F 75 20 70 6C 65 61 73 65 20 65 6E 74 65 72 20 32 20 6E 75 6D 62 65 72 73 3A 20 0A 00 25 64 20 25 64 00 00 00 00 00 00 00 0A 25 73 20 74 68 65 20 61 64 64 69 74 69 6F 6E 20 6F 66 20 74 77 6F 20 6E 75 6D 62 65 72 73 20 69 73 20 3A 20 25 64 0A 00 00 00 00 00 00 00 00 50 72 65 73 73 20 74 68 65 20 5B 45 6E 74 65 72 5D 20 6B 65 79 20 74 6F 20 63 6F 6E 74 69 6E 74 75 65 2E 2E 2E 00 00 00 40 75 40 00 00 00 00 00 60 70 40 00 00 00 00 00 30 19 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 A0 40 00 00 00 00 00 08 A0 40 00 00 00 00 00 FC 75 40 00 00 00 00 00 40 90 40 00 00 00 00 00	Please enter your name: .%s.....Hello %s. Can you please enter 2 numbers: ..%d %d........%s the addition of two numbers is : %d.........Pre ss the [Enter] key to contintue..... .@u@.....`p@.. ...0.@........ .... @.... .. @.....üu@.. ...@.@.....	

Assembly (.asm)

Code: "Example1.asm"

```
	.file	"main.c"
	.text
	.def	__main;	.scl	2;	.type	32;	.endef
	.section .rdata,"dr"
.LC0:
	.ascii "Please enter your name: \0"
.LC1:
	.ascii "%s\0"
	.align 8
.LC2:
	.ascii "Hello %s. Can you please enter 2 numbers: \12\0"
.LC3:
	.ascii "%d %d\0"
	.align 8
.LC4:
	.ascii "\12%s the addition of two numbers is : %d\12\0"
	.align 8
.LC5:
	.ascii "Press the [Enter] key to continue...\0"
	.text
	.globl main
	.def	main;	.scl	2;	.type	32;	.endef
	.seh_proc	main
main:
	pushq	%rbp
	.seh_pushreg %rbp
	movq	%rsp, %rbp
	.seh_setframe%rbp, 0
	addq	$-128, %rsp
	.seh_stackalloc	128
	.seh_endprologue
	movl	%ecx, 16(%rbp)
```

```
        movq    %rdx, 24(%rbp)
        call    __main
        leaq    .LC0(%rip), %rcx
        call    printf
        leaq    -80(%rbp), %rax
        movq    %rax, %rdx
        leaq    .LC1(%rip), %rcx
        call    scanf
        leaq    -80(%rbp), %rax
        movq    %rax, %rdx
        leaq    .LC2(%rip), %rcx
        call    printf
        leaq    -88(%rbp), %rdx
        leaq    -84(%rbp), %rax
        movq    %rdx, %r8
        movq    %rax, %rdx
        leaq    .LC3(%rip), %rcx
        call    scanf
        movl    -88(%rbp), %edx
        movl    -84(%rbp), %eax
        movl    %eax, %ecx
        call    sums
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %edx
        leaq    -80(%rbp), %rax
        movl    %edx, %r8d
        movq    %rax, %rdx
        leaq    .LC4(%rip), %rcx
        call    printf
        leaq    .LC5(%rip), %rcx
        call    printf
        call    getch
        movl    $0, %eax
        subq    $-128, %rsp
        popq    %rbp
        ret
        .seh_endproc
        .globl sums
        .def    sums;  .scl    2;      .type  32;     .endef
        .seh_proc     sums
sums:
        pushq  %rbp
        .seh_pushreg %rbp
        movq    %rsp, %rbp
        .seh_setframe%rbp, 0
        subq    $16, %rsp
        .seh_stackalloc     16
        .seh_endprologue
        movl    %ecx, 16(%rbp)
        movl    %edx, 24(%rbp)
        movl    16(%rbp), %edx
        movl    24(%rbp), %eax
        addl    %edx, %eax
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        addq    $16, %rsp
        popq    %rbp
        ret
        .seh_endproc
```

```
        .ident "GCC: (tdm64-1) 9.2.0"
        .def   printf;      .scl   2;     .type  32;    .endef
        .def   scanf; .scl   2;     .type  32;    .endef
        .def   getch; .scl   2;     .type  32;    .endef
```

C (.c)

Code: “Example1.c”

```
//------------------------------------------------------------------------------
// Name:        Example1.c
// Purpose:     Example
//
// Platform:    Win64, Ubuntu64
//
// Author:      Axle
// Created:     16/12/2021
// Updated:     18/02/2022
// Copyright:   (c) Axle 2021
// Licence:     MIT No Attribution
//------------------------------------------------------------------------------

// Single line comment
/* Multi-line
comment */

#include <stdio.h>  // Include standard library headers.
#include <stdlib.h>  // An inline comment.

int sums(int num1, int num2);  // Declare the function name.
#define MAX_STRING_SIZE 64  // MAX 64 Characters string length.

int main(int argc, char *argv[])  // Main procedure.
    {
    char user_name[MAX_STRING_SIZE];  // Create a string variable.
    int num1, num2, return_value;  // Create Integer Variables.

    printf("Please enter your name: ");  // Print text to console.
    scanf("%s", user_name);  // Get user input.
    printf("Hello %s. Can you please enter 2 numbers:\n", user_name);
    printf("?");
    scanf("%d", &num1);
    printf("?");
    scanf("%d", &num2);
    // v Call our function and send the 2 Integer variables to it.
    return_value = sums(num1, num2);
    // v Print the returned results.
    printf("\n%s the addition of two numbers is : %d\n", user_name, return_value);

    printf("Press the [Enter] key to continue...");
    getchar();  // Pause the program until a key is pressed
    return 0;
    }

int sums(int num1, int num2)  // Function to add 2 numbers
    {
    int num3;
    num3 = num1 + num2;
```

```
    return num3;  // Return the results to the calling statement
    }
```

FreeBASIC (.bas)

Code: “Example1.bas”

```
'--------------------------------------------------------------------------------
' Name:         Example1.bas
' Purpose:      Example
'
' Platform:     Win64, Ubuntu64
'
' Author:       Axle
'
' Created:      16/12/2021
' Updated:       18/02/2022
' Copyright:    (c) Axle 2021
' Licence:      MIT No Attribution
'--------------------------------------------------------------------------------

Rem FreeBASIC hello World
' Single line comment
/' Multiline
comment '/

Declare Function main_procedure() As Integer
Declare Function sums(num1 As Integer, num2 As Integer) As Integer

main_procedure()

Function main_procedure() As Integer  ' main procedure
    Dim user_name As String  ' Create a string variable.
    Dim As Integer num1, num2, return_value  ' Create Integer Variables.

    Input "Please enter your name: ", user_name ' Print text to console, and get
user input.
    Print "Hello " & user_name & ".";
    Print " Can you please enter 2 numbers:"
    Input num1  ' Get user input.
    Input num2  ' Get user input.
    return_value = sums(num1, num2)  ' Call our function and send the 2 Integer
variables to it.
    Print user_name & " the addition of two numbers is : " & return_value

    Sleep  ' Sleep until a key is pressed.
    Return 0
End Function

' Function to add 2 numbers.
Function sums(num1 As Integer, num2 As Integer) As Integer
    Dim num3 As Integer
    num3 = num1 + num2
    Return num3  ' Return the results to the calling statement.
End Function
```

Python 3 (.py)

Code: "Example1.py"

```
#-------------------------------------------------------------------------
# Name:        Example1.c
# Purpose:     Example
#
# Platform:    Win64, Ubuntu64
#
# Author:      Axle
# Created:     16/12/2021
# Updated:     18/02/2022
# Copyright:   (c) Axle 2021
# Licence:     MIT No Attribution
#-------------------------------------------------------------------------
# Standard headers (Modules) are included by default

# Single line comment
## Highlighted comment

def main(): # Main procedure
    user_name: str  # Create a string variable.
    num1: int; num2: int; return_value: int # Create Integer Variables.

    print("Please enter your name: ", end='')  # Print text to console.
    user_name = str(input())  # Get user input
    print("Hello ", user_name, end=' ')
    print("Can you please enter 2 numbers:")
    num1 = int(input("?"))
    num2 = int(input("?"))
    return_value = sums(num1, num2)  # Call our funtion and send the 2 Integer
variables to it.
    print(user_name, " the addition of two numbers is : ", return_value)

    input("press [Enter] to continue...")  # Pause the program until a key is
pressed.
    return None

def sums(num1, num2):  # Function to add 2 numbers.
    num3: int
    num3 = num1 + num2
    return int(num3)  # Return the results to the calling statement.

if __name__ == '__main__':
    main()
```

Likely the most noticeable difference will be the user-defined Function named **sums()** where we add the two values supplied by the user and return a result.

I have also included a special function named **main()** or **main_procedure()**.

Main() is like the manager of the application and controls all the program flow for the application. You have to imagine the complexity of an application with 5000 lines of code and many functions! The other parts are defining Variables and User Input and Output. Again, don't try to understand what is really happening in the application for now; we can come back to it when we have finished this booklet ☺

Which programming language to choose?

There are hundreds of programming languages in use across the world of computing and this choice is one of the most difficult and confusing tasks that any new programmer will be faced with.

The reality is that there is no one size fits all correct answer to this question.

All programming languages have strengths and weaknesses - some are more conducive to mathematical tasks while other languages may be stronger at creating GUIs or better suited to creating websites.

A programming language choice will most often be based on our desired application, in that web design may need knowledge of HTML and JavaScript, whereas OS designers may use C or C++, and a database programmer would need to learn SQL.

Most programmers will learn many programming languages over time, and choose four to eight favourite languages for most programming tasks, such as HTML, JavaScript, C++, SQL, etc.

Due to the sheer size of the programming needed in a significant software development project, coding is most often done in a team environment with many contributing developers, committing their code to a version control system. Members of the team may specialize in a programming language, or certain project need, and each member will contribute a section of code to the overall code base and main software project. I will discuss a little more about modular programming in the heading “Additional components”.

Every software development team will establish a set of programming languages and tools based on the required outcome(s) of the project. A new employee or member to that team may need to quickly become familiar with programming languages not earlier used, and this may involve a steep learning curve.

Programmers that have learned the fundamental aspects of programming structure will find most languages familiar based on those core coding principles, and should pick up on a new programming language quickly. Students who have skipped the basics to rush into learning a high level language in an attempt to obtain a quick, one step, ‘lots of bling outcome’, may struggle to integrate into a new programming environment.

Students starting with a simplified programming environment that covers the core principles of coding and software design will progress more rapidly later on.

A small self-contained scripting language such as Scratch, AutoIt, and even traditional languages, such as C will help to develop those basic skills.

Keep it simple and focus on using core programming techniques, such as creating small code blocks to solve simple mathematical equations, simple user input, simple results printed to a screen, practicing decisions and repetition, and the use of functions to recycle repeated code.

As boring as simple text-based applications may seem, they rapidly teach us the essential coding skills needed in more complex projects. Jumping ahead and trying to create fancy bling GUIs will quickly land a new programmer in a world of frustration and confusion!

C-based languages are the basis for most industrial programming, and learning a C-based language will help in familiarization with about 70% of programming languages in use in the commercial

programming world. Most programming languages are written in C and often carry many of the C language attributes, look, and feel. Java, VBScript, JavaScript, AutoIt, Scratch, etc. all use a similar syntax and have a familiar look and feel about them.

A common term students may encounter when starting programming languages is BASIC.

BASIC is an acronym for **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode.

Many articles may claim that BASIC is a legacy programming language that is no longer in use, and is an inaccurate and misleading claim.

Many variations of Traditional BASIC programming languages are still in popular use, including in the creation of industrial applications.

FreeBASIC is a worthwhile example of a modern implementation of BASIC. FreeBASIC uses the traditional language keywords and structure from MS QBASIC and Quick BASIC as well as some other BASIC languages. It also includes principles that are unique to FreeBASIC. FreeBASIC, unlike most of its predecessors, pre-compiles into C source and ultimately is assembled into native machine language instead of being run through an interpreter as a script. FreeBASIC is also available for both Windows and Linux based systems.

I am not intending to promote FreeBASIC as a production language here. I included it as it offers a sound bridge between historical language constructs and modern languages such as Python.

BASIC also refers to the style of human readable code and coding structure introduced in BASIC programming. The BASIC programming coding style has played a strong influence in the creation of most of the modern programming languages in use today. Descriptions of many coding languages will often include a "…has an easy to read 'BASIC' like syntax…".

Visual-type programming languages, such as Scratch may seem easier to use, and do play an important role in the early development of programming skills, but may also hinder a student's progress if used for too long.

Most programming in industry is done in a text-based editor, writing real code line by line. Even though these text-based code lines are displayed within the coloured blocks in teaching aids, such as Scratch, moving into a text-based programming environment at an early stage will help the students' understanding of programming.

A quick note on AutoIt

AutoIt is a Scripting language based on C, C++, and the Windows API, which is in essence a programming library for the C Language. AutoIt is a simplification of the C programming language and the Windows programming library. It is a good tool for beginners because of its simplicity, sound help documents, emphasis on sound use of programming fundamentals, as well as its similarity to C/C++.

AutoIt can make use of most C/C++ language abilities, as well as full use of Windows programming libraries, making it capable of creating commercial/Industrial strength applications. AutoIt also offers an easy introduction to C/C++/C# languages.

AutoIt is only available for Windows-based devices.

So what about all that code?

You might ask, 'I looked at some source code and there are thousands of lines, it looks so confusing?'

Yes, it can seem confusing at first, but if we follow the fundamental coding principles and start with small applications it will all quickly begin to make sense.

Most programs are created with smaller blocks of code and we only need to focus on each block when we are writing an application. Each block fits together and plays its part in the larger application and is generally all linked together in the main program block.

Think of a house which is a fairly complex structure and traditionally built one block at a time. Fortunately, like building modern houses which are often built with pre-assembled walls, kitchens, roofs, etc. modern programming comes with pre-built blocks of code that we can quickly assemble into a complex program with just a few lines of code.

To give an example, early machine language coders needed to write some 20,000 lines of code just to create a basic graphic window on a computer! Later languages such as C need about 50 lines of code to create a window using the Windows built-in code library, and modern languages can create a window with just one line of code!

The 20,000 lines of code are still used by the computer and called from a pre-written code Library. Libraries are large collections of well created and tested code designed in such a way that we can just use one or two lines of code in our program to make use of the thousands of lines of complex coding in the background. To put it another way, well trained software engineers have done all the hard work for us, we just have to follow a few basic rules and glue together the parts we need with a few well written lines of code.

We don't usually see the library code in our program, but our application makes use of it when it is running. Most of the bulk in size of an OS install, such as Windows, is pre-created code libraries used in creating the applications used in that system. Many of these libraries are licensed in such a way that they are also available to software developers creating software for the platform.

It's worth noting that compiled machine code is often still usable in much the same way as textual source code. Most programming libraries will come as both text-based source code and pre-compiled machine code. Downloadable applications such as Igor Pavlov's' 7-Zip, NirSoft's dll_export, Angus Johnson's Resource_Hacker and MiTeC's EXE_Explorer can go a long way toward illustrating the contents of compressed executable files including the blocks of machine code. The names of usable function blocks in DLLs are written in plain text within the blocks of compiled byte code. Executable files are really just compressed archives of computer code blocks.

Industry use

When people enter the realm of programming it is easy to think of programming in terms of personal computers, mobile devices and web platforms. As important as these are, they only make up part of the story. Industry, manufacturing and everyday objects in our world are heavily reliant upon both software and firmware programming. Knowledge of programming in an industrial environment is a beneficial asset to have as it opens many opportunities in what may be an over saturated world of Web and App developers.

Creating software and even firmware for industry applications does not require any exceptional departure in skill from creating Apps for personal computer platforms. We have many easy to access development boards such as PIC, Arduino and Raspberry Pi that we can practice and test ourselves on. Most of these System-On-Chip (SOC) and Single-board computers (SBC) platforms can be programmed using the tools that I have used throughout this booklet. Most hardware will use C almost predominantly with other languages such as Blockly Sketch and Micro Python as some of the higher level alternatives.

Understanding the fundamentals of programming opens up the world of working with industry hardware and applications as seamlessly as it does for web and personal device based apps.

Automation
Objects in our everyday life are powered by software and firmware. A modern automobile will easily have 6 to 10 small computer systems controlling everything from the engine and driveline to the air conditioner to multimedia and even the locking of the doors. Each of these systems is required to be programmed at some point in its development. Even modern automotive technicians are required to have some knowledge of programming to interact with, diagnose and correct faults in these systems.

Most factories are driven almost exclusively by automated computer systems. Every step of a production line will be controlled by a networked computer system from a central master computer down to smaller programmable slave controlers. All of these systems require an initial setup as well as support staff to replace, repair and update the software and firmware that powers them.

There is almost no place in our modern lives that is not driven by computer programming.

IoT
IoT is another facet of industry where all manner of devices are becoming both interconnected as well as connecting via the internet. One of the common SBCs as mentioned above is the Espressif range of WiFi boards, with the ESP 8266 and ESP32 being common players in WiFi IoT. These boards are often programmed in much the same way as other development boards and even Arduino has SBCs based upon the ESP SOC range. These boards are small and can be readily connected to all manner of devices including other development boards to allow a mixture of communication and remote control for almost anything imaginable. They are programmed in the same way as any other programming environment and the firmware is uploaded to the chip directly from the IDE.

Robotics
Robotic devices exist all about us, from the home toaster that pops out our toast, to industrial robots with large arms helping to assemble automobiles on a production line.

Often Robotics is controlled by some form of AI. This AI is not really referring to Artificial Intelligence, but more commonly to an automated controller Interface.

All AI is actually software written by a programmer carrying out a predefined set of tasks based on pre-programmed decisions calculated from data received by the application. Small robotic teaching aids, such as Sphero can help students to understand the programmers' role in industrial and commercial automation as well as simple popup help dialogs when we visit a web page.

AI
Most often AI is misrepresented as what is really industry automation, although it has come to ecompase a broader term of applications. True AI is more often referred to as "Machine Learning", where the computer system is capable of creating new concepts without intervention from the

programmer. For example a neural network that can use computer vision and speech recognition to learn about its environment, learn a spoken language and begin to make inferences and decision based upon the data it receives. In time it may begin to create images and structured speech that is not a direct reflection of the data it has consumed.

Aerospace
In recent years the world has began a renewed push in space exploration from Mars exploration to amitions to revisit the moon. Space travel is extremely complex and landing a rover on Mars some 300 million kilometers away is no small feat. To make this even more challenging we can't directly control the device as there is a 10 to 40 minute round trip for a radio signal to Mars. Programming is the key component for the success of these missions. Even flying a service vehicle at close range from earth to the ISS is a complex task that requires many years of training and multiple personnel to achieve. Modern aerospace can now fly a craft to the ISS and back without a pilot and relying entirely apon software and firmware driven systems.

Summary

Coding, programming, and software engineering are traditionally quite complex subjects and is the realm of computer engineers. That complexity is even more prevalent in today's IT world due to the sheer size of the industry and large number of programming paradigms in common use. It can be a daunting and even frightening first step for any person entering into that world.

Educational tools, such as Scratch and Sphero go a long way toward making that first step a little less frightening. Teaching programming fundamentals, good coding practices, combined with the use of education tools can open pathways to a student that may otherwise feel that it is beyond their reach.

Easy to learn, difficult to master. Like any learned skill the first steps in programming are not overly difficult to grasp. Then, as the students' skills grow, so too does the range of more complex concepts that become available to them. It is a step by step growth process that repeatedly cycles between skill learning and skill application in a "Spiralled Cycle of Learning".

I feel that it is important for us as parents and educators to emphasize the importance of a sound understanding of computer fundamentals, programming fundamentals, and sound coding practices.

It is far too easy with modern programming tools to focus on the end result of software design and neglect what is occurring behind the scene.

A student and programmer with a greater awareness of what is occurring behind the scene will be more skilled at writing fast, effective, and more competitive software. Missing these important understandings runs the risk of creating a world of irresponsible coders and broken software, and places the Australian job market in a less competitive position into the future.

Objectives of this booklet

The main objective of this booklet is to offer the reader an overall concept of the programming methodology common to most mainstream programming languages.

This should give the reader a greater capacity to look at (read) any source code from a common language, to be able to identify the core components, as well as have some understanding of the programmatic flow of the application. My hope is that you will gain a capacity to have a general understanding of any common programming language by relating the common methodology to the specific keywords of the language, thus making the learning curve of any language a more simplified task.

This booklet is not intended to teach any single language but rather offer some exposure to the essential concepts of programming.

I will go into learning the actual languages in my other booklets. View this as a primer for the more targeted booklets.

This booklet works in conjunction with the 2nd booklet in the series "A BEGINNERS GUIDE TO PROGRAMMING – Book 2- Development Environment Overview" which explains the neccissary steps to install and make use the IDEs, compilers and development environment used for the examples in this booklet. This will allow you to gain experience with some aid of the examples as well as introduce the reader into 3 of the common programming development platforms.

Sometimes our first impression of programming can feel like walking into a dark room and stumbling around because we not only don't know where the light switches are, we don't even know what they look like or what to look for! Hopefully I can describe some of the common light switches so you can find them more quickly.

Note, each step in the learning process has a way of inherently illuminating the light switches for higher levels… As the learning process continues a student will begin to notice more of the higher level concepts, almost as if some magical light switch has been turned on.

"Give a man a fish and you feed him for a day; teach a man to fish and you feed him for a lifetime." Unknown

---

A brief history of computers

Computers have really been in existence for thousands of years! How so? The principles of a computer system are not strictly confined to an electronic device.

The Abacus is an historical example of a kinetic based computer, and the Babylonians created systems for solving complex mathematical problems which are related to the programming algorithms we use in modern electronic computing. Try working out the square root of any number based upon a set of strict step by step rules where each step of the calculation is shown.

Another common computer system is that of the analogue computer, often implemented as a Fluid Based Computer. Analog and Fluid based computers are kinetic based and don't need electronics.

Although this sounds like something from the industrial revolution, fluid-based computers are extremely common in modern industry.

Livestock watering troughs on outback stations implement a float which controls the water level in the drinking trough. This same implementation can be found in our home water closets, where a float device senses when to fill the holding tank and when to stop the flow of water once the tank is full. This is an example of an analogue computer switch monitoring the amount of fluid in the system and controlling the fluid levels according to a predefined set of rules.

Traditional motor vehicle automatic transmissions are complex fluid computers needing no electronic controls. Automatic transmissions use an intricate maze of fluid channels, sensing pressures and vehicle speeds, changing to the correct gear as needed. Automated industry production lines as well as many industries needing some form of robotic assistance use this same “analog computing” intensively.

Analog computers are traditionally more robust and can be capable of far more precise calculations than logic based electronic computers. They are also expensive and generally quite bulky in size, as well as costly to maintain. Modern industry is progressively converting many of these systems to electronic-based computer controls, or more correctly a combination of analogue and electronic.

It is in the interests of a student looking toward a career in programming or software engineering to have an awareness of the history of computing, and of analogue computing, as it does form a large part of the industry.

Computer fundamentals

- ## What is a computer?

  A computer is a physical and electronic device that accepts information from somewhere and manipulates that information according to a set of rules (commands) and returns the new information to somewhere.

  The essential components of a computer related to basic software development are the central processing unit (CPU) usually called the processor, random access memory (RAM) or sometimes called the memory, input and output (I/O, such as keyboard, monitor, storage devices, etc.), storage, such as solid state drive (SSD) hard disk drive (HDD), flash drive, etc., and other peripherals, such as printers and WiFi.

  RAM is most important to the programmer as all code instructions, programs, and data, information, must exist in the computer’s memory before being sent to the CPU for manipulation. It is returned from the CPU to RAM after each task is carried out. We can think of RAM as the marshalling yard for the CPU.

- ## What is an OS?

  An operating system (OS) is a special program that allows communication between humans and the computer hardware. It is a mix of both source code and binary code. We can think of

it as a language interpreter that converts human language and concepts to computer language.

The OS looks after many hidden and background tasks and helps to make our use of computers a simpler task. The OS also has a number of helper applications that help it to do its work.

The OS also gives a programming interface allowing us to use the pre-existing software and code that is shipped with the OS. This allows us to write short pieces of code that re-use the thousands of lines of code that already exist in the OS as well as its helper libraries. We can create a GUI Window in one line of code which would otherwise need thousands of lines of code to create.

- ## What is a user interface?

A user Interface (UI) are the visual components we see when using a computer. The UI, Desktop, Explorer windows, or any visual display should not be confused with the OS as they are entirely separate entities even though they work together as a team.

The UI is responsible for giving us a comfortable and familiar way of interacting with the computer's OS. The UI also offers a large amount of reusable code base allowing us to create professional UIs for our software without the need to rewrite all that code.

- ## What is the difference between source code and byte code?

Source code is the human readable form of a software application (text), whereas byte code is the machine-readable version of a software application (hexadecimal).

Most source code is converted into byte code as it is smaller to transport and faster for the computer to run in its native language. You can tell this by the application's name, such as *MyApp.exe*. It is also commonplace to execute/run source code directly. An application converts it into a machine-readable form "on the fly", such as your web browser loading a file *MyWebPage.html*. Source code is larger to transport and slower to run, but has the benefit of being human-readable, easier to maintain, or change. Source code driven applications are often referred to as scripting languages.

In the examples given earlier I provided a small sample of the byte code which is the lowest level or closest to the hardware.

The next is a low level language known as **Assembly**. Assembly code relates directly and in detail to the computer hardware. Writing Assembly is a slow, precise, and tedious task compared to high level languages.

The next example in C is a medium level language and makes use of a high degree of "Abstraction" from the hardware. Simple user-relatable command statements can perform complex tasks with less coding. C Languages are known as *compiled languages* as they are first compiled and optimised for a variety of different hardware, and then assembled into very fast running and small sized *machine code*.

FreeBASIC is a compiled language the same as C, and the source code is really converted into C-language source code before being compiled. Traditionally BASIC languages are interpreted, meaning they are always run from source code and converted into machine executable code “on the fly” by a scripting engine or script interpreter.

Python is an *interpreted* language and is always run from source code. Python is first compiled to what is called *bytecode.* So, unlike C/C++, which translates source code to machine code, this python bytecode is a low-level set of instructions that can be executed by an interpreter. Instead of executing instructions on your CPU, bytecode instructions are executed on a virtual machine, making Python platform independent. Python can also be converted into C/C++ source code that can be compiled and assembled to machine executable code, although python is most commonly used as a scripted language.

- ## What is the programmer's role in computing?

The programmer’s role is to facilitate the interaction between a computer user and the computer hardware, as well as interaction with other devices, such as robotics or automation.

In essence the programmer's role is to issue all the required commands to the CPU to guide the input, output, and manipulation of data (information) in a meaningful way. This also includes the design of UIs that offer a familiar way for users to interact with your software.

It is also good practice for programmers to be mindful of the amount of resources used in an application that they write. All applications need a share of system RAM, as well as a share of CPU cycles, and it is the programmer’s responsibility to use these resources in a responsible way. This comes down to experience and sound coding practices as your skills develop.

Programming is most commonly performed within an Integrated Development Environment or IDE, which gathers all the required programming resources together into one place to facilitate easier development of software. This will include one, or sometimes a number of, the hundreds of programming languages in common use, resource libraries, such as images, pre-written blocks of code, audio snips, etc., and a code editor. Most will include tools to test (debug) the code, as well as run the application so we can see our progress. Many IDEs will also include applications that will convert our code into bytecode, machine code, and portable executables.

A portable executable (PE) is a compressed archive containing all the machine code, data, and resources that an application needs to “Run”. When the *.exe file is run, the various components are unpacked from the .exe archive into the computer's RAM. The OS is then pointed to the “Entry Point”, a memory location, that is the first executable line of code in the application. This entry point loosely corresponds to the entry point of our source code at the top of the page source, or **main().**

Portable executables, or PE files, can be easily opened to view the contents with an application such as 7-Zip. Although we cannot directly edit the packed executable file, we can see the basic contents of the PE file format. Specialised tools are also available to inspect the contents of PE files.

Programming fundamentals

The 8 core aspects of coding

1. **Keywords** - if, else, while, for, next, Func, Local, etc.
2. **Operators** - Mathematical and logic symbols/operators. Assignment, Concatenation, Mathematical, Comparison, Bitwise, Logical, Conditional.
3. **Comments** - REM ; // /**/ /--><--/ #CS #CE, etc.
4. **Variables** - Variables are a named placeholder to store information.
5. **Statements, expressions, and procedures** – The core commands that give instructions of what to do as well as calculations. Often referring to a line or group of commands.
6. **Decisions** - If Then Else, Select Case, Switch Case, Ternary.
7. **Loops** - Loop Until, For Next, While WEnd, Do Until, Do While, For In Next, Goto.
8. **Functions** - Reusable blocks of code, Function name, Gosub, Subroutine.
9. **Additional components**

Keywords

if, else, while, for, next, Func, Local, etc.

Keywords make up our BASE set of commands used by a programming language - usually about 16-40 words. All other significant commands are built from the base set of Keywords.

All languages implement their own set of keywords, although in time the programmer will begin to see that they all follow the same basic set of rules and will appear similar. A little like the difference between the spelling of AUS Colour and USA Color. It can be a source of confusion, as unlike humans that can read past spelling mistakes, computers require us to be “Syntax” perfect.

The following is a basic list of the keywords and Operators from C, FreeBASIC, and Python 3. The keywords, operators, as well as some tokens and symbols are essentially “The Programming Language”.

Most of the additional words we encounter at first are library functions that have been created from the keywords. When we learn a programming language we are learning the core components that make up the language. Learning different libraries is an extended or more advanced component of becoming a programmer, and is separate from the language, although many languages will come with a “Standard” library associated with it.

C language Keywords							
auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

BASIC language Keywords	
Operators See Operator List	I If...Then

	IIf
.	ImageConvertRow
...	ImageCreate
	ImageDestroy
_	ImageInfo
__DATE__	Imp
__DATE_ISO__	Implements
__FB_64BIT__	Import
__FB_ARG_COUNT__	Inkey
__FB_ARG_EXTRACT__	Inp
__FB_ARG_LEFTOF__	Input (Statement)
__FB_ARG_RIGHTOF__	Input (File I/O)
__FB_ARGC__	Input #
__FB_ARGV__	Input()
__FB_ARM__	InStr
__FB_ASM__	InStrRev
__FB_BACKEND__	Int
__FB_BIGENDIAN__	Integer
__FB_BUILD_DATE__	Is (Select Case)
__FB_BUILD_DATE_ISO__	Is (Run-Time Type Information Operator)
__FB_BUILD_SHA1__	IsDate
__FB_CYGWIN__	IsRedirected
__FB_DARWIN__	
__FB_DEBUG__	K
__FB_DOS__	Kill
__FB_ERR__	
__FB_EVAL__	L
__FB_FPMODE__	LBound
__FB_FPU__	LCase
__FB_FREEBSD__	Left
__FB_GCC__	Len
__FB_GUI__	Let
__FB_JOIN__	Lib
__FB_LANG__	Line
__FB_LINUX__	Line Input
__FB_MAIN__	Line Input #
__FB_MIN_VERSION__	LoByte
__FB_MT__	LOC
__FB_NETBSD__	Local
__FB_OPENBSD__	Locate
__FB_OPTION_BYVAL__	Lock
__FB_OPTION_DYNAMIC__	LOF
__FB_OPTION_ESCAPE__	Log
__FB_OPTION_EXPLICIT__	Long
__FB_OPTION_GOSUB__	LongInt
__FB_OPTION_PRIVATE__	Loop
__FB_OUT_DLL__	LoWord
__FB_OUT_EXE__	LPos
__FB_OUT_LIB__	LPrint
__FB_OUT_OBJ__	LSet
__FB_PCOS__	LTrim

__FB_QUOTE__
__FB_SIGNATURE__
__FB_SSE__
__FB_UNIQUEID__
__FB_UNIQUEID_POP__
__FB_UNIQUEID_PUSH__
__FB_UNIX__
__FB_UNQUOTE__
__FB_VECTORIZE__
__FB_VER_MAJOR__
__FB_VER_MINOR__
__FB_VER_PATCH__
__FB_VERSION__
__FB_WIN32__
__Fb_X86__
__FB_XBOX__
__FILE__
__FILE_NQ__
__FUNCTION__
__FUNCTION_NQ__
__LINE__
__PATH__
__TIME__

#
#assert
#define
#else
#elseif
#endif
#endmacro
#error
#if
#ifdef
#ifndef
#inclib
#include
#lang
#libpath
#line
#macro
#pragma
#print
#undef

$
$Dynamic
$Include
$Lang
$Static

M
Mid (Statement)
Mid (Function)
Minute
MKD
MkDir
MKI
MKL
MKLongInt
MKS
MKShort
Mod
Month
MonthName
MultiKey
MutexCreate
MutexDestroy
MutexLock
MutexUnlock

N
Naked
Name
Namespace
New (Expression)
New (Placement)
Next
Next (Resume)
Not
Now

O
Object
Oct
OffsetOf
On Error
On...Gosub
On...Goto
Once
Open
Open Com
Open Cons
Open Err
Open Lpt
Open Pipe
Open Scrn
Operator
Option()
Option Base
Option ByVal

?	Option Dynamic
? (Shortcut For 'Print')	Option Escape
? # (Shortcut For 'Print #')	Option Explicit
? Using (Shortcut For 'Print Using')	Option Gosub
	Option Nogosub
A	Option NoKeyword
Abs	Option Private
Abstract (Member)	Option Static
Access	Or
Acos	Or (Graphics Put)
Add (Graphics Put)	OrElse
Alias (Name)	Out
Alias (Modifier)	Output
Allocate	Overload
Alpha (Graphics Put)	Override
And	
AndAlso	P
And (Graphics Put)	Paint
Any	Palette
Append	pascal
As	PCopy
Asc	Peek
Asin	PMap
Asm	Point
Assert	PointCoord
AssertWarn	Pointer
Atan2	Poke
Atn	Pos
	Preserve
B	PReset
Base (Initialization)	Print
Base (Member Access)	Print #
Beep	Print Using
Bin	Private
Binary	Private: (Access Control)
Bit	ProcPtr
BitReset	Property
BitSet	Protected: (Access Control)
BLoad	Pset (Statement)
Boolean	Pset (Graphics Put)
BSave	Ptr (Shortcut For 'Pointer')
Byref (Parameters)	Public
Byref (Function Results)	Public: (Access Control)
Byref (Variables)	Put (Graphics)
Byte	Put # (File I/O)
ByVal	
	R
C	Random
Call	Randomize
CAllocate	Read
Case	Read (File Access)

Cast	Read Write (File Access)
CBool	Reallocate
CByte	ReDim
CDbl	Rem
cdecl	Reset
Chain	Restore
ChDir	Resume
Chr	Resume Next
CInt	Return (From Procedure)
Circle	Return (From Gosub)
Class	RGB
Clear	RGBA
CLng	Right
CLngInt	RmDir
Close	Rnd
Cls	RSet
Color	RTrim
Command	Run
Common	
CondBroadcast	S
CondCreate	SAdd
CondDestroy	Scope
CondSignal	Screen
CondWait	Screen (Console)
Const	ScreenCopy
Const (Member)	ScreenControl
Const (Qualifier)	ScreenEvent
Constructor	ScreenGLProc
Constructor (Module)	ScreenInfo
Continue	ScreenList
Cos	ScreenLock
CPtr	ScreenPtr
CShort	ScreenRes
CSign	ScreenSet
CSng	ScreenSync
CsrLin	ScreenUnlock
CUByte	Second
CUInt	Seek (Statement)
CULng	Seek (Function)
CULngInt	Select Case
CUnsg	SetDate
CurDir	SetEnviron
CUShort	SetMouse
Custom (Graphics Put)	SetTime
Cva_Arg	Sgn
Cva_Copy	Shared
Cva_End	Shell
Cva_List	Shl
Cva_Start	Short
CVD	Shr
CVI	Sin

CVL
CVLongInt
CVS
CVShort

D
Data
Date
DateAdd
DateDiff
DatePart
DateSerial
DateValue
Day
Deallocate
Declare
DefByte
DefDbl
defined
DefInt
DefLng
DefLongInt
DefShort
DefSng
DefStr
DefUByte
DefUInt
Defulongint
DefUShort
Delete (Statement)
Destructor
Destructor (Module)
Dim
Dir
Do
Do...Loop
Double
Draw
Draw String
DyLibFree
DyLibLoad
DyLibSymbol

E
Else
ElseIf
Encoding
End (Block)
End (Statement)
End If
Enum
Single
SizeOf
Sleep
Space
Spc
Sqr
Static
Static (Member)
stdcall
Step
Stick
Stop
Str
Strig
String (Function)
String
StrPtr
Sub
Sub (Member)
Sub (Pointer)
Swap
System

T
Tab
Tan
Then
This
Thiscall
ThreadCall
ThreadCreate
ThreadDetach
ThreadSelf
ThreadWait
Time
Timer
TimeSerial
TimeValue
To
Trans (Graphics Put)
Trim
True
Type (Alias)
Type (Temporary)
Type (Udt)
TypeOf

U
UBound
UByte
UCase

GetJoystick GetKey GetMouse GoSub Goto H Hex HiByte HiWord Hour	Xor Xor (Graphics Put) Y Year Z ZString

Python 3 Language Keywords				
False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Operators

Mathematical and logic symbols/operators. Assignment, Concatenation, Mathematical, Comparison, Bitwise, Logical, and Conditional.

+ - / * = < > ?, etc., all have significant meanings as well as double meanings depending upon the **context** in which they are used.

For example, "**X=5**" assigns a value of **5** to the variable **X** on the left hand side, whereas "**if X=5 then**" tests for truth to see if variable X is equal to 5. It should also be noted that "**if X=5 then**" is the same statement as "**if X==5 then**", but not the same as "**X=5**". You can see from this example why context is important.

Operators control both the data manipulation as well as the logical flow of an application and are used in conjunction with the other fundamental programmatic components. In reality a computer (CPU) is just a fast manipulator of mathematical data, and everything else is just a higher level abstraction.

C Language Operators		
Category	**Operator**	**Associativity**
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right

Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

FreeBASIC Operators

Assignment Operators

- =[>] (Assignment)
- &= (Concatenate And Assign)
- += (Add And Assign)
- -= (Subtract And Assign)
- *= (Multiply And Assign)
- /= (Divide And Assign)
- \= (Integer Divide And Assign)
- ^= (Exponentiate And Assign)
- Mod= (Modulus And Assign)
- And= (Conjunction And Assign)
- Eqv= (Equivalence And Assign)
- Imp= (Implication And Assign)
- Or= (Inclusive Disjunction And Assign)
- Xor= (Exclusive Disjunction And Assign)
- Shl= (Shift Left And Assign)
- Shr= (Shift Right And Assign)
- Let (Assignment)
- Let() (Assignment)

Type Cast Operators

- Cast (Operator)
- CPtr

Arithmetic Operators

- + (Add)
- - (Subtract)
- * (Multiply)
- / (Divide)
- \ (Integer Divide)
- ^ (Exponentiate)

Relational Operators

- = (Equal)
- <> (Not Equal)
- < (Less Than)
- <= (Less Than Or Equal)
- >= (Greater Than Or Equal)
- > (Greater Than)

Bitwise Operators

- And (Conjunction)
- Eqv (Equivalence)
- Imp (Implication)
- Not (Complement)
- Or (Inclusive Disjunction)
- Xor (Exclusive Disjunction)

Short Circuit Operators

- Andalso (Short Circuit Conjunction)
- Orelse (Short Circuit Inclusive Disjunction)

Preprocessor Operators

- # (Argument Stringize)
- ## (Argument Concatenation)
- ! (Escaped String Literal)
- $ (Non-Escaped String Literal)

Pointer Operators

- @ (Address Of)

- Mod (Modulus)
- - (Negate)
- Shl (Shift Left)
- Shr (Shift Right)

Indexing Operators

- () (Array Index)
- [] (String Index)
- [] (Pointer Index)

String Operators

- + (String Concatenation)
- & (String Concatenation With Conversion)
- Strptr (String Pointer)

- * (Value Of)
- Varptr (Variable Pointer)
- Procptr (Procedure Pointer)

Type or Class Operators

- . (Member Access)
- -> (Pointer To Member Access)
- Is (Run-Time Type Information Operator)

Memory Operators

- New Expression
  - New Overload
- Placement New
- Delete Statement
  - Delete Overload

Iteration Operators

- For, Next, and Step

Python 3 Language Operators		
Arithmetic operators		
Operator	**Meaning**	**Example**
+	Add two operands or unary plus	x + y+ 2
-	Subtract right operand from the left or unary minus	x - y- 2
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y)
//	Floor division - division that results into a whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y (x to the power y)
Comparison operators		
Operator	**Meaning**	**Example**
>	Greater than - True if left operand is greater than the right	x > y
<	Less than - True if left operand is less than the right	x < y
==	Equal to - True if both operands are equal	x == y
!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to - True if left operand is greater than or equal to	x >= y

	the right	
<=	Less than or equal to - True if left operand is less than or equal to the right	x <= y
Logical operators		
Operator	**Meaning**	**Example**
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x
Bitwise operators		
Operator	**Meaning**	**Example**
&	Bitwise AND	x & y = 0 (0000 0000)
|	Bitwise OR	x | y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x >> 2 = 2 (0000 0010)
<<	Bitwise left shift	x << 2 = 40 (0010 1000)
Assignment operators		
Operator	**Example**	**Equivalent to**
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5
Special operators		
Operator	**Meaning**	**Example**

is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True
Operator	**Meaning**	**Example**
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Comments

*REM ; // /**/ /--><--/ #CS #CE, etc.*

Comments and REMarks are a very important part of programming and should not be ignored. Comments are not actively used by the application, but are there as a guide for the programmer or any other person who may read your code in the future. It is not uncommon to have thousands of lines of code in an application. Commenting our source code makes it easier to understand what a block of code is for and what other parts of a program it is related to.
In short, it is best practice to briefly comment on all blocks of code and significant statements.

C language Comments/Remarks	
/*…*/	Single line comment
//… (>C99)	Single line comment
/* … */	Block Comment

FreeBasic language Comments/Remarks	
Rem …	Single line Comment
‘ …	Single line comment
/’…’/	In line and Block comment

Python 3 language Comments/Remarks	
# …	Single line comment
# … # … # …	Block comment
## …	Highlighted comment (Editor specific)
“”” … “””	Single line documentation string (not intended for comments)
“”” … “””	Multi line documentation string (not intended for comments)

Variables

Variables are a named placeholder to store information.

Variables are a named placeholder to store information (data). This also includes *arrays* and other higher level data structures and objects. Traditionally there are two types of variables (data types), *Integer* and *Character Strings*.

Integers are just numbers, 0-9, and characters are single-text letters and numbers derived from the ANSI/ASCII printable character set of 96 characters. An integer number 7 is **not** the same as the text letter '7', although some programming languages will allow integers and text numbers to be used interchangeably.

An easy way to think of variables is like an envelope with your name on it, the variable name, and some money inside as the data or value allocated to the variable name, such as a variable named "Phone_Credit" contains a value of "30" dollars.

We can view the contents of the variable "Phone_Credit" and retrieve the content value of 30, or we could use the money and subtract 30 from the contents of our variable.

Pseudo Code:

```
Var_ Phone_Credit (Create the envelope and give it a useful name)
Var_ Phone_Credit = 30 (Put 30 into the envelope)
Print(Var_ Phone_Credit) (Print the value of Var_ Phone_Credit to the screen)
>>30
Var_ Phone_Credit = Var_ Phone_Credit – 30 (Remove 30 from the envelope to buy
credit)
 (0 = 30 – 30 , The right side of the equation is always calculated before the
left of the = operator)
Print(Var_ Phone_Credit) (Print the value of Var_ Phone_Credit to the screen)
>>0
```

Statements, expressions, and procedures

The core commands that give instructions of what to do. Often referring to a line or group of commands.

Statement vs Expressions vs Procedures

- **Statements** represent an action or command
- **Expressions** are a combination of variables, operations and values that yields a result
- **Procedures** are a group of statements and expressions that perform an overall task.

Procedures, Routines, Functions

Multiple expressions/lines of code performing multiple tasks to achieve an outcome are often referred to as routines. Routines are often reusable or repeated blocks of code, often occurring within a Function block. We will cover more about modular programming in the section on Functions and Additional Components.

Code/Pseudo Code: (REM is a Remark or Comment)

```
REM Phone credit balance check Procedure
Var_Ph_Credit_Balance = Var_Phone_Credit - Var_PhCredit_Puchase REM Expression

IF Var_Ph_Credit_Balance < 0 REM Conditional Statement
Print "Alert! You don't have enough phone credit balance" REM Statement
```

Decisions

If Then Else, Select Case, Switch Case, Ternary

Decisions control the flow of the application and redirect the program execution to different components within our main programs code structure.

All source code is organized into "Blocks" of reusable code and decisions redirect the program execution to the block of code we wish to make use of. Decisions are like the logic, thinking, or intelligent part of the application.

Code/Pseudo Code:

```
REM Phone credit balance check Procedure
Var_Ph_Credit_Balance = Var_Phone_Credit - Var_PhCredit_Puchase REM Expression

IF Var_Ph_Credit_Balance < 0 THEN REM Conditional Statement or Decision
    PRINT "Alert! You don't have enough phone credit balance." REM Statement
ELSE IF Var_Ph_Credit_Balance > 0 THEN REM Conditional Statement or Decision
    PRINT "You have more than enough phone credit balance." REM Statement
ELSE IF Var_Ph_Credit_Balance == 0 THEN REM Conditional Statement or Decision
    PRINT "You have enough phone credit balance, but should top up soon." REM
Statement
ELSE
    PRINT "Could not make a choice or there was an error."
```

Loops

Loop Until, For Next, While WEnd, Do Until, Do While, For In Next, Goto

The greatest strength of a computer is its ability to carry out complex tasks in a repetitive way.

Repetition is the core strength of a computer and of programming. This is not unlike creating a complex Word document template. We do the hard work and create it once, and then reuse the template many times adding in our own little pieces of information each time without the need to rewrite the entire document each time.

Goto is an important programming command, but unfortunately it has been removed from many languages due to its misuse.

In the past, programming languages didn't have while loops, if statements, etc., and in the early days of C, programmers often coming from an assembly background, would use goto to create incredibly hard-to-understand code - programmers used GOTO to make up the 'logic' of their programs. This gave us code that was almost impossible to keep maintained. You might have heard the term "spaghetti code". Using GOTO often led to code that was unnecessarily difficult to read, understand, and maintain - spaghetti code.

Most of the time, we live without goto and are fine. There are a few instances, however, where goto can be useful. Using a goto to jump out of a deeply-nested loop can often be cleaner than using a condition variable and checking it on every level. Generally nowadays, we use methods, conditionals, and loops.

In a later section I will give an example of what GOTO LABEL: is and why it exists.

Code/Pseudo Code:

```
REM Phone credit active call balance monitoring loop
Var_Cost_Per_Minute = 0.68

While((Var_Ph_Credit_Balance > 0) Or (Call_End == True))
   Var_Ph_Credit_Balance = Var_Ph_Credit_Balance – Var_Cost_Per_Minute
   Sleep(60) REM Time in seconds.
End While

IF Var_Ph_Credit_Balance < 0 THEN
    PRINT "Alert! You don't have enough phone credit balance."
ELSE IF Var_Ph_Credit_Balance > 0 THEN
    PRINT "You have more than enough phone credit balance."
ELSE IF Var_Ph_Credit_Balance == 0 THEN
    PRINT "You have enough phone credit balance, but should top up soon."
ELSE
    PRINT "Could not make a choice or there was an error."
```

Functions

Reusable Blocks of code, Function name, Gosub, Subroutine

Functions are reusable "Blocks" of code. Write it once and re-use it as many times as needed, often billions of re-uses per second!

Functions are usually sent a set of values as Parameters/Arguments and Return a new value.

Objects and classes are similar to functions but are capable of a greater ability to communicate with a programmers' main code as well as the ability to communicate with other Objects.

Code/Pseudo Code:

```
FUNCTION Main(Arguments) REM Start of a Function
…
GOSUB Phone_Credit_Check()
…
RETURN 0
END Main REM End of a Function

REM Phone credit balance check Subroutine
SUB Phone_Credit_Check() REM Start of a Subroutine
Var_Ph_Credit_Balance = Var_Phone_Credit - Var_PhCredit_Puchase REM Expression

IF Var_Ph_Credit_Balance < 0 THEN REM Conditional Statement or Decision
    PRINT "Alert! You don't have enough phone credit balance." REM Statement
ELSE IF Var_Ph_Credit_Balance > 0 THEN REM Conditional Statement or Decision
    PRINT "You have more than enough phone credit balance." REM Statement
ELSE IF Var_Ph_Credit_Balance == 0 THEN REM Conditional Statement or Decision
    PRINT "You have enough phone credit balance, but should top up soon." REM
Statement
ELSE
    PRINT "Could not make a choice or there was an error."

END Phone_Credit_Check REM End of a Subroutine
```

```
REM Next Function, Subroutine…
```

Code/Pseudo Code:

```
FUNCTION Main() REM Start of a Function
…
Phone_Credit_Check()
…
END Main REM End of a Function

REM Phone credit balance check Function
FUNCTION Phone_Credit_Check() REM Start of a Function
Var_Error_Level
Var_Ph_Credit_Balance = Var_Phone_Credit - Var_PhCredit_Puchase REM Expression

IF Var_Ph_Credit_Balance < 0 THEN REM Conditional Statement or Decision
    PRINT "Alert! You don't have enough phone credit balance." REM Statement
    Var_Error_Level = 0
ELSE IF Var_Ph_Credit_Balance > 0 THEN REM Conditional Statement or Decision
    PRINT "You have more than enough phone credit balance." REM Statement
    Var_Error_Level = 0
ELSE IF Var_Ph_Credit_Balance == 0 THEN REM Conditional Statement or Decision
    PRINT "You have enough phone credit balance, but should top up soon." REM
Statement
    Var_Error_Level = 0
ELSE
    PRINT "Could not make a choice or there was an error."
    Var_Error_Level = 1

RETURN Var_Error_Level
END Phone_Credit_Check REM End of a Function

REM Next Function…
```

Additional components

There are many additional components that make up a programming language. The above subset of core components is common to most programming languages and allows for a strong base from which you can progress to more complex ideas.

Styles.

Coding styles can vary from language to language and over time we have a natural tendency toward developing our own coding style. There are general guides and standards that exist in most programming languages and it is important to make an effort towards following the coding styles that are set out. Styles can be ambiguous and even contradictory when working across multiple languages. The important thing is to attempt to be consistent especially within the boundaries of an individual coding project.

Libraries (Modules).

One of the most important additions are “Libraries”. Libraries are large collections of well tested, well proven, reusable code blocks. Libraries are created from the programming language keywords and in some cases even from other libraries.

Instead of writing many thousands of lines of source code every time we start a new project, we can use the premade code from libraries. Libraries (Collections) will come in many categories such as GUI libraries, graphics libraries, math libraries, game creation libraries, and so on. These libraries have been created by many programmers and engineers that have done all of the hard coding work, and most often we just need to add a few lines of code to call complex procedures from the library.

Static vs Runtime libraries.

Static libraries are included (or merged) into the application source code before the application is transferred to RAM and executed. In compiled portable executables that are assembled into the final .exe, the static library is included into the final .exe. Static libraries will always be allocated a share of RAM as part of the running **.exe**. When multiple applications use the same library, each application creates its own duplicate of the library in its runtime space in RAM.

Runtime libraries, or dynamically linked libraries (DLL), are compiled into a .dll, or in the case of a Linux system, a “Shared Object” **.so**. Runtime library modules, DLLs, are called from our application only when needed during the application's execution, and are released from RAM when no longer needed. Many applications can call the same DLL at runtime, removing the need to transport the entire library with each application. The OS manages how each application gets a share of the DLLs resources and time allocation.

Each method has pros and cons, and comes down to the programmer’s choice to meet the requirements of the final application.

Cross platform coding.

All examples of source code are written in such a way that they are “Cross Platform” and will compile or execute on both Windows and Ubuntu 64 bit versions. In most cases the lines of source code will be the same for both systems if we are staying strictly within the “STD Library”. The std library is *standard* across linux and Unix systems, but is very limited. It does not take long until we need to make use of OS dependent requests from the Operating System. In this case a different function call will be required depending upon the OS that the source is used on at the time. Although I have attempted to limit the use of OS dependent source, in some cases it is unavoidable. At the top of the page as well as amidst some routines you will encounter If OS a, Else OS b to direct the program flow to the correct line for the OS. I will explain the use of libraries, and cross platform coding in more detail in the section “Additional Components” toward the end of the booklet.

Debugging.

Debugging is the art of finding and correcting errors within our code and code that does not produce the outcome that we expect. There are a number of tools and methods for finding errors and tracing the expected output of our application. It is important to make every effort to create error free applications as well as applications that produce the result that is required in a consistent way.

Exception Handling.

Exception handling is the practice of dealing with unpredictable data inputs into our application. There are many instances where a file may not exist or a user may enter data that is outside of the expectations of our software. As programmers and coders we are required to “Predict” any occasion where this could occur and create “Boundaries” that capture and handle program flow if the data goes outside of the expected boundaries.

Code safety and security.

Writing secure and safe code is critical in importance once we move into production coding. We have all heard terms like “Zero Day”, “Exploit” and “Flood”. Writing secure code can be challenging, but there are simple steps and methods that will lead to secure code outcomes.

Core language examples

The main application flow

The **Main** Procedure is the main management centre for an application. Main Procedure is the boss directing and delegating the tasks for the rest of the application. The boss isn’t interested in doing all the trivial work of the employees and subcontractors.

The main procedure looks after the main outline and program flow of the application. All tasks are delegated to worker Functions and Subroutines. Sometimes, in larger applications, the boss may delegate the decision making to another function, known as a *Callback Procedure*, but we won’t be going into that in this booklet.

In the following code examples I have shown the essential outline of an application including the “Main Procedure''.

Execution of the application begins at the top, first line, of the source code, and progresses until it reaches the last line, which is the default exit point when the application closes.

The top of the page is called the *program entry point*, but some languages can use, or will require a “Formal Entry Point” somewhere after the first line in your source code. In C languages, the formal entry point is a function called **main()**, *which is mandatory*.

In scripting languages, such as BASIC and Python, the main procedure can be the entire source code, or a formal Main Procedure. In Python, the “Formal Entry Point'' is the first “If” statement that is encountered. Although it is common practice to exclude the **main()** function in many scripting languages, I *do* recommend using a formal “Main Procedure ''.

Execution starts at the top of your code and ends at the last line of your code.

The order in which certain tasks are carried out is important to the “Program Flow''. **Main()** keeps all the important application control in a neatly encapsulated function and can never become

accidentally mixed up with other code routines. A Main procedure also allows us to keep a separation between “Global” data and “Local” data, but more on data “Scope” later.

I will use the **main_procedure()** format shown in the following examples throughout all of the code examples in this document.

"Everything should be made as simple as possible, but no simpler." - Albert Einstein

Language: C (.c)

Code example: “Main_procudure.c”

```c
// Start Of File - Application Entry Point
// Module description
// Library includes
// Global Definitions
// Function Definitions

int main(int argc, char *argv[])  // Main procedure – “Formal Entry Point”
    {
    // Local Definitions
    // Code routines

    return 0
    }

// End Of File - Application exit
```

Language: FreeBASIC

Code example: “Main_procudure.bas”

```
' Start Of File - Application entry
' Module description
' Library includes
' Global Definitions
' Function Definitions

Function main_procedure() As Integer  ' Main procedure – “Formal Entry Point”
    ' Local Definitions
    ' Code routines

    Return 0
End Function

' End Of File - Application exit
```

Language: Python 3

Code example: “Main_procudure.py”

```python
# Start Of File - Application Entry Point
# Module description
# Library includes
# Global Definitions

def main():  # Main procedure
    # Local Definitions
    # Code routines
```

```
    return None

# Function Definitions

if __name__ == '__main__':  # “Formal Entry Point”
    main()

# End Of File - Application Exit
```

In the following illustration I have shown the basic principle of program flow and decision making.

The application starts, creates two variables, introduces the guessing game, and then asks the user to input a number guess between 0 and 10.

In the following diamond shaped boxes the application does several comparison tests against the contained value of the variable “Number_To_Guess” against the value of “User_Guess”. If the comparison is true, it issues a notice to the user and returns to the user input box in a “Loop”. If the comparison is false it moves down to the next decision. The application will continue to loop back and ask the user to enter a number until it is false for all of the comparisons (i.e. User_Guess == 8). At that point the application exits the forever loop, issues a congratulation, and then exits the program.

The following flowchart forms part of the programming design and logic and is independent of the language used.

A more detailed explanation of application design and flowchart is covered in the section “Additional Components”.

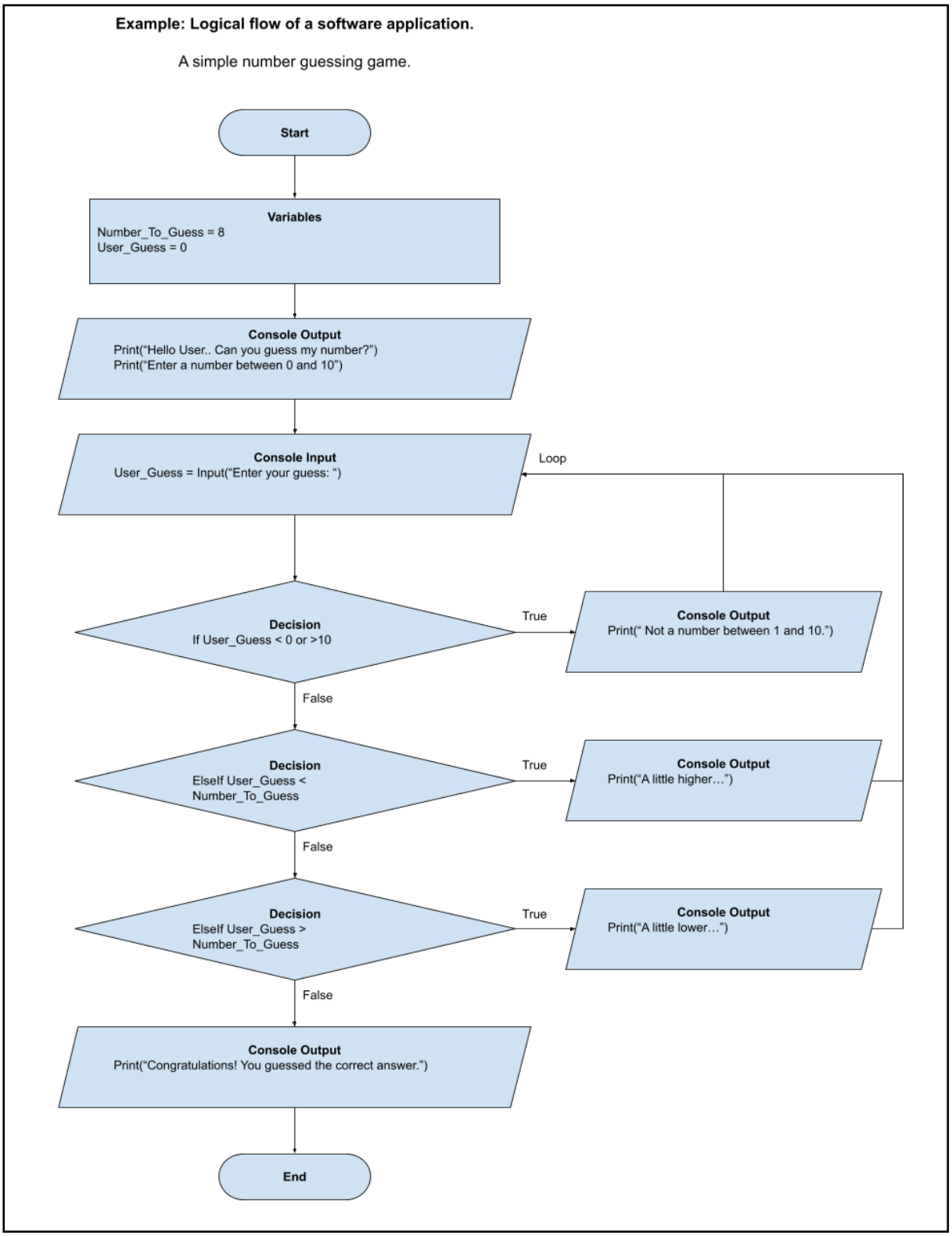


Illustration of program flow.

In the following code examples the **while loop** controls the main logic and program flow. There can be many loops within an application, but the **main_procedure()** will generally contain one significant "while" loop that will continue to repeat until such time the application ends. In a small application such as the examples following, it is fine to include some Input and print statements and even a little

mathematical expression, but once we begin to create even slightly more complex applications the boss will need to begin delegating the tasks out to some of the worker “Functions” or “Subroutines”.

C Example of program flow control from the flow chart

Code example: “Guess.c”

```
//------------------------------------------------------------------------------
// Name:        guess.c
// Purpose:     Guess a number
//
// Platform:    Win64, Ubuntu64
//
// Author:      Axle
//
// Created:     18/12/2021
// Updated:     19/02/2022
// Copyright:   (c) Axle 2021
// Licence:     MIT No Attribution
//------------------------------------------------------------------------------

#include <stdio.h>  // include standard library headers
//#include <stdlib.h>

int main(int argc, char *argv[])  // Main procedure
    {
    int Number_To_Guess = 8;  // Create our variables.
    int User_Guess = 0;  // Create our variables.

    printf("Hello User.. Can you guess my number?\n");
    printf("Enter a number between 1 and 10\n");

    while(1)  // Loop forever or until a 'Break" statement is reached.
        {
        printf("Enter your guess: ");
        scanf("%d", &User_Guess);  // Get the user input from the console.
        // v Conditional statement using || Logical Or
        if ((User_Guess < 1) || (User_Guess > 10))
            {
            printf("Not a number between 1 and 10!\n");
            }
        else if (User_Guess < Number_To_Guess)  // Conditional statement
            {
            printf("A little higher...\n");
            }
        else if (User_Guess > Number_To_Guess)  // Conditional statement
            {
            printf("A little Lower...\n");
            }
        else  // If nothing else found
            {
            break;  // Breaks out of the while loop (Logically the same as == 8)
            }
        }

    printf("Congratulation! You guessed the number.\n");

    printf("Press the [Enter] key to continue...");
```

```
    getchar();  // Pause the program until a key is pressed
    return 0;
    }
```

In the FreeBASIC example you will see that I have declared the **main_procedure()** function two times. Many programming languages require that all functions are declared to exist *before* they are called and used. This means that all additional functions in the application must be placed at the start of your code, before the formal entry point, aka **main_procedure()**. Most languages let us get around this by declaring just the first line of the function and placing the full function and routines later in the code below our **main_procedure()**.

In the FreeBASIC example following, I have called **main_procedure()** before the actual **Function**, so I have placed a declaration (`Declare Function main_procedure() As Integer`) in the "Header" of the code.

FreeBASIC Example of program flow control from the flow chart

Code example: "Guess.bas"

```
'------------------------------------------------------------------------------
' Name:         guess.bas
' Purpose:      FreeBASIC Guess a number
'
' Platform:     Win64, Ubuntu64
'
' Author:       Axle
'
' Created:      18/12/2021
' Updated:      19/02/2022
' Copyright:    (c) Axle 2021
' Licence:      MIT No Attribution
'------------------------------------------------------------------------------

Declare Function main_procedure() As Integer

main_procedure()

Function main_procedure() As Integer  ' Main procedure
    Dim Number_To_Guess As Integer = 8  ' Create our variables.
    Dim User_Guess As Integer  ' Create our variables.

    Print "Hello User... Can you guess my number?"
    Print "please enter a number between 1 and 10"

    While (1)  ' Loop forever or until an "Exit While" statement is reached.
        Input "enter your guess:", User_Guess  ' Get the user input from the
console.

        If (User_Guess < 1) Or (User_Guess > 10) Then  ' Conditional statement
using Logical "Or".
            Print "Not a number between 1 and 10"
        Elseif (User_Guess < Number_To_Guess) Then  ' Conditional statement.
            Print "A little higher..."
        Elseif (User_Guess > Number_To_Guess) Then  ' Conditional statement.
            Print "A little Lower..."
        Else  ' If nothing else found.
            Exit While  '  Breaks out of the while loop (Logicly the same as = 8).
```

```
        End If

    Wend

    Print "Congratulations! you guessed the correct answer."


    Print "Press any key to continue..."
    Sleep  ' Sleep until a key is pressed
    Return 0
End Function
```

Python also requires that all functions be known before the formal entry point and before the first "If" condition statement. We can place the functions below **main()** by placing an if statement and call to main() as the last line in the application.

```
if __name__ == '__main__':
    main()
```

This forces all functions and declarations to be read before formerly starting the application in **main()**

Python 3 Example of program flow control from the flow chart

Code: "Guess.py"

```
#-------------------------------------------------------------------------------
# Name:         guess.py
# Purpose:      Guess a number
#
# Platform:     REPL, Win64, Ubuntu64
#
# Author:       Axle
#
# Created:      18/12/2021
# Updated:      19/02/2022
# Copyright:    (c) Axle 2021
# Licence:      MIT No Attribution
#-------------------------------------------------------------------------------

# Standard headers (Modules) are included by default

def main():  # Main procedure
    Number_To_Guess: int = 8  # Create our variables.
    User_Guess: int  # Create our variables.

    print("Hello User... Can you guess my number?")
    print("Enter a number between 1 and 10")

    while 1:  # Loop forever or until a 'Break" statement is reached.
        User_Guess = int(input("Enter your guess: "))  # Get the user input from
the console.

        if (User_Guess < 1) or (User_Guess > 10):  # Conditional statement using
Logical or.
            print("Not a number between 1 and 10!")
        elif (User_Guess < Number_To_Guess):  # Conditional statement.
            print("A little higher...")
```

```
        elif (User_Guess > Number_To_Guess):  # Conditional statement.
            print("A little lower...")
        else:  # If nothing else found
            break  # Breaks out of the while loop (Logicly the same as = 8).

    print("Congratulation! You guessed the number.")

    input("press [Enter] to continue...")  # Pause the program until a key is
pressed.
    return None

if __name__ == '__main__':
    main()
```

Keywords

if, else, while, for, next, Func, Local, etc.

Keywords make up our BASE set of commands used by a programming language, usually about 16-40 words. All other significant commands are built from the base set of Keywords.

From the earlier examples you will now be noticing the similarities and differences between the keywords of the three languages I am using. The following is a table with some of the common keywords from each language, highlighting the similarities and differences. The important takeaway is that they all perform a similar task in programming, and the underlying methods , principles, and structure remain the same between each language.

C	FreeBASIC	Python 3	Description
//, /*...*/	Rem, ‘	#	Comments or Remarks.
printf()	Print	Print()	Prints characters to the console screen.
Scanf()	Input	Input()	Gets user input from the console.
while	While	while	Loop with flow control logic.
for	For	for	Loop with iteration with counting.
if	If … Then	if	Control flow statement for conditional branching.
else if	ElseIf	elif	Control flow statement testing for multiple conditions
switch case	Select Case	match case	Control flow statement for conditional branching. Similar to “If true/false Then” and ElseIf but can select multiple matching comparisons at the same time.
break	Exit While	break	Break out of a while loop regardless of whether the while control condition has been met or not. In the earlier examples, **while (1)** is the same as **while (true)**. It can never be resolved and will create an “endless loop”, so I have used a break if at least one condition is met. I could have used “while (`User_Guess != not-equal-to Number_To_Guess`) instead of the else statement.

#include	#include	import	Include contents of another source file. Mostly used for source files from libraries. The file(s) are included at the top of the page and read as though they are part of the source file you are working on. This adds to the description about functions always being at the top of your source code.

Functions	**Description**
int sum(int num1, int num2);	C Declaration of a Function.
return_value = sum(4, 7)	C Calling the function and assigning the return value to the variable “return_value”.
Function sum(num1 As integer, num2 As integer) As Integer	FreeBASIC Declaration of a Function.
return_value = sum(4, 7)	FreeBASIC Calling the function and assigning the return value to the variable “return_value”.
def sums(num1, num2):	Python 3 Declaration of a Function.
return_value = sums(4, 7)	Python 3 Calling the function and assigning the return value to the variable “return_value”.

Operators

Mathematical and logic symbols/operators. Assignment, Concatenation, Mathematical, Comparison, Bitwise, Logical, Conditional

The main category of Operators

- Arithmetic Operators
- Increment and Decrement Operators
- Assignment Operators
- Relational Operators
- Conditional operators
- Logical Operators
- Bitwise Operators

Operator Precedence and order of operations

Operator Precedence and order of operations can be one of the more confusing and challenging aspects of programming. Even when we follow what we believe to be normal mathematical rules we may still run into instances where the results are not as we expected.

Every programming language will have its own Order of Operations set out in a table under *Operator Precedence*. Always check the table if you are encountering erroneous results.

Mathematical rules or mnemonics, such as BODMAS/PEDMAS will not follow 100% true in a programming environment. BODMAS may be better described as BO(DM)(AS) as Division and Multiplication have no precedence and are calculated from left to right, and are the same for Addition and Subtraction. Assignments and some other operators are ordered from right to left.

If you are uncertain, or want to be sure, we can force precedence by the use of parentheses 1+2*3=7, (1+2)*3=9.

Different languages will have a slightly different set of Operators although many will be common between languages.

C Language Operators		
Highest Precedence from top down		
Category	**Operator**	**Associativity**
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

FreeBASIC Operators		
Highest Precedence from top down		
Operator	Description	Associativity
CAST	Type Conversion	N/A
PROCPTR	Procedure pointer	N/A
STRPTR	String pointer	N/A
VARPTR	Variable pointer	N/A
[]	String index	Left-to-Right
[]	Pointer index	Left-to-Right
()	Array index	Left-to-Right
()	Function Call	Left-to-Right
.	Member access	Left-to-Right
->	Pointer to member access	Left-to-Right
@	Address of	Right-to-Left

*	Value of	Right-to-Left
New	Allocate Memory	Right-to-Left
Delete	Deallocate Memory	Right-to-Left
^	Exponentiate	Left-to-Right
-	Negate	Right-to-Left
*	Multiply	Left-to-Right
/	Divide	Left-to-Right
\	Integer divide	Left-to-Right
MOD	Modulus	Left-to-Right
SHL	Shift left	Left-to-Right
SHR	Shift right	Left-to-Right
+	Add	Left-to-Right
-	Subtract	Left-to-Right
&	String concatenation	Left-to-Right
Is	Run-time type information check	N/A
=	Equal	Left-to-Right
<>	Not equal	Left-to-Right
<	Less than	Left-to-Right
<=	Less than or equal	Left-to-Right
>=	Greater than or equal	Left-to-Right
>	Greater than	Left-to-Right
NOT	Complement	Right-to-Left
AND	Conjunction	Left-to-Right
OR	Inclusive Disjunction	Left-to-Right
EQV	Equivalence	Left-to-Right
IMP	Implication	Left-to-Right
XOR	Exclusive Disjunction	Left-to-Right
ANDALSO	Short Circuit Conjunction	Left-to-Right
ORELSE	Short Circuit Inclusive Disjunction	Left-to-Right

=[>]	Assignment	N/A
&=	Concatenate and Assign	N/A
+=	Add and Assign	N/A
-=	Subtract and Assign	N/A
*=	Multiply and Assign	N/A
/=	Divide and Assign	N/A
\=	Integer Divide and Assign	N/A
^=	Exponentiate and Assign	N/A
MOD=	Modulus and Assign	N/A
AND=	Conjunction and Assign	N/A
EQV=	Equivalence and Assign	N/A
IMP=	Implication and Assign	N/A
OR=	Inclusive Disjunction and Assign	N/A
XOR=	Exclusive Disjunction and Assign	N/A
SHL=	Shift Left and Assign	N/A
SHR=	Shift Right and Assign	N/A
LET	Assignment	N/A
LET()	Assignment	N/A

Python 3	
Highest Precedence from top down	
Operator	**Description**
(expressions...), [expressions...], {key: value...}, {expressions...}	Binding or parenthesized expression, list display, dictionary display, set display
x[index], x[index:index], x(arguments...), x.attribute	Subscription, slicing, call, attribute reference
await x	Await expression
**	Exponentiation
+x, -x, ~x	Positive, negative, bitwise NOT
*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, remainder
+, -	Addition and subtraction
<<, >>	Shifts
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparisons, including membership tests and identity tests

not x	Boolean NOT
and	Boolean AND
or	Boolean OR
if – else	Conditional expression
lambda	Lambda expression
:=	Assignment expression

Read more about Operator Precedence at Rosetta Code.

C Example – Basic use of operators

Code: "Operator.c"

```
//------------------------------------------------------------------------------
// Name:        Operators.c
// Purpose:     Example
//
// Platform:    Win64, Ubuntu64
//
// Author:      Axle
// Created:     20/12/2021
// Updated:     19/02/2022
// Copyright:   (c) Axle 2021
// Licence:     MIT No Attribution
//------------------------------------------------------------------------------

#include <stdio.h>  // include standard library headers
#include <stdlib.h>

int main(int argc, char *argv[])  // Main procedure
    {
    int num1 = 4;  // Assignment Operator.
    int num2 = 2;  // Assignment Operator.

    num1 = (num1 + num2) / 2;  // Assignment '=', Arithmetic '+' and '/', 3 = (4 +
2) / 2
    num1 += 2;  // Assignment Operator. Same as num1 = num + 2
    num1 -= 2;  // Assignment Operator. 4 = 6 - 2
    num2++;  // Increment Operator. num2 = num2 + 1
    num2--;  // Decrement Operator. num2 = num2 - 1

    if (num1 == num2)  // Equality Operator
        {
        printf("num1 and num2 contain the same value.\n");
        }
    else if (num1 != num2)  // Inequality Operator
        {
        printf("num1 and num2 do not contain the same value.\n");
        }
    else
        {
        printf("The 2 Equality tests above will never allow the program to reach
here.\n");
        }

    if (num1 < num2)  // Relational Operator
```

```
        {
        printf("num1 is less than num2.\n");
        }
    else if (num1 > num2)  // Relational Operator
        {
        printf("num1 is greater than num2.\n");
        }
    else
        {
        printf("The only option left is that they must be equal.\n");
        }

    if ((num1 < 10) && (num2 < 10))  // Logical AND '&&' and Relational Operators
'<'
        {
        printf("The value of both variables is less than 10.\n");
        }
    else if ((num1 > 2) || (num2 > 2))  // Logical OR '||' and Relational
Operators '<'
        {
        printf("The value of at least 1 of the variables is greater than 2.\n");
        }
    else
        {
        printf("Neither of the above tests were True and no decision was
made.\n");
        }

    if ((num1 * num2) < 10)  // Compound Arithmetic and relational expression
inside of an if statement
        {
        printf("The product of num1 and num2 is less than 10.\n");
        }

    printf("Press the [Enter] key to continue...");
    getchar();  // Pause the program until a key is pressed
    return 0;
    }
```

FreeBASIC Example – Basic use of operators

Code: "Operator.bas"

```
'--------------------------------------------------------------------------------
' Name:         Operators.bas
' Purpose:
'
' Platform:     Win64, Ubuntu64
'
' Author:       Axle
' Created:      20/12/2021
' Updated:      19/02/2022
' Copyright:    (c) Axle 2021
' Licence:      MIT No Attribution
'--------------------------------------------------------------------------------

Declare Function main_procedure() As Integer
```

```
main_procedure()

Function main_procedure() As Integer  ' Main procedure
    Dim num1 As Integer = 4  ' Assignment Operator
    Dim num2 As Integer = 2  ' Assignment Operator


    num1 = (num1 + num2) / 2  ' Assignment '=', Arithmetic '*' and '/', 3 = (4 +
2) / 2
    num1 += 2  ' Assignment Operator. Same as num1 = num + 2
    num1 -= 2  ' Assignment Operator. 4 = 6 - 2
    num2 += 1  ' FreeBASIC has no Increment Operator. Use variable Assignment += 1
    num2 -= 1  ' FreeBASIC has no Decrement Operator. Use variable Assignment -= 1

    If (num1 = num2) Then  ' Equality Operator
        Print "num1 and num2 contain the same value."
    Elseif (num1 <> num2) Then  ' Inequality Operator
        Print "num1 and num2 do not contain the same value."
    Else
        Print "The 2 Equality tests above will never allow the program to reach
here."
    End If

    If (num1 < num2) Then  ' Relational Operator
        Print "num1 is less than num2."
    Elseif (num1 > num2) Then  ' Relational Operator
        Print "num1 is greater than num2."
    Else
        Print "The only option left is that they must be equal."
    End If

    If ((num1 < 10) And (num2 < 10)) Then  ' Logical AND '&&' and Relational
Operators '<'
        Print "The value of both variables is less than 10."
    Elseif ((num1 > 2) Or (num2 > 2)) Then  ' Logical OR '||' and Relational
Operators '<'
        Print "The value of at least 1 of the variables is greater than 2."
    Else
        Print "Neither of the above tests were True and no decision was made."
    End If

    If ((num1 * num2) < 10) Then  ' Compound Arithmetic and relational expression
inside of an if statement
        Print "The product of num1 and num2 is less than 10."
    End If

    Print "Press any key to continue..."
    Sleep  ' Sleep until a key is pressed
    Return 0
End Function
```

Python 3 Example – Basic use of operators
Code: “Operator.py”
#--- # Name: Operators.py # Purpose:

```
#
# Platform:    REPL, Win64, Ubuntu64
#
# Author:      Axle
# Created:     20/12/2021
# Updated:     19/02/2022
# Copyright:   (c) Axle 2021
# Licence:     MIT No Attribution
#-------------------------------------------------------------------------------

def main():
    num1: int = 4  # Assignment Operator
    num2: int = 2  # Assignment Operator

    num1 = (int(num1) + int(num2)) / 2  # Assignment '=', Arithmetic '*' and '/',
3 = (4 + 2) / 2
    num1 += 2  # Assignment Operator. Same as num1 = num + 2
    num1 -= 2  # Assignment Operator. 4 = 6 - 2
    num2 += 1  # Python 3 has no Increment Operator. Use variable Assignment += 1
    num2 -= 1  # Python 3 has no Decrement Operator. Use variable Assignment -= 1

    if (num1 == num2):  # Equality Operator
        print("num1 and num2 contain the same value.")
    elif (num1 != num2):  # Inequality Operator
        print("num1 and num2 do not contain the same value.")
    else:
        print("The 2 Equality tests above will never allow the program to reach
here.")

    if (num1 < num2):  # Relational Operator
        print("num1 is less than num2.")
    elif (num1 > num2):  # Relational Operator
        print("num1 is greater than num2.")
    else:
        print("The only option left is that they must be equal.")


    if ((num1 < 10) and (num2 < 10)): # Logical AND '&&' and Relational Operators
'<'
        print("The value of both variables is less than 10.")
    elif ((num1 > 2) or (num2 > 2)): # Logical OR '||' and Relational Operators
'<'
        print("The value of at least 1 of the variables is greater than 2.")
    else:
        print("Neither of the above test were True and no decision was made.")


    if ((num1 * num2) < 10): # Compound Arithmetic and relational expression
inside of an if statement
        print("The product of num1 and num2 is less than 10.")


if __name__ == '__main__':
    main()
```

Comments

*REM ; // /**/ /--><--/ #CS #CE, etc.*

Try to use brief and meaningful comments in your source code.

Imagine you want another coder, or perhaps yourself in 12 months' time, to easily see what your application is doing. Name or explain obscure code routines and especially what a function is meant to achieve and return.

I have over commented the following code a little as I wanted to explain some simple methods that I have used.

C Example – Commented Source Code

Code: "Comments.c"

```c
//------------------------------------------------------------------------------
// Name:        C99 to C17 Comments example
// Purpose:     Illustrate Comments with a simple math Function
// Requires:    <stdio.h>, <stdlib.h>
// Usage:       Read
//
//------------------------------------------------------------------------------
// Author:      Axle
// Copyright:   (c) Axle 2021
// Licence:     MIT No Attribution
// Created:     21/12/2021
// Modified:
// Versioning   ("MAJOR.MINOR.PATCH") (Semantic Versioning 2.0.0)
// Script V:    0.0.2 (alpha)
//              Alpha is functional, but missing features; Beta Is functional
//              but may still contain bugs; Release is fully functional and
//              bug free)
// Encoding:    ANSI
// Compiler V:  TDM-GCC 9.2.0 32/64-bit
// OS Scope:    (Windows)
// UI Scope:    (CLI)
//------------------------------------------------------------------------------
// NOTES:
//
//
//------------------------------------------------------------------------------

// ---> START Library Imports
#include <stdio.h>
#include <stdlib.h>
// END Library Imports <---
// ---> START Global Defines
double SquareRoot(double Radicand);
// END Global Defines <---

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

    printf("Enter a number to find the Square Root: ");
    scanf("%lf", &User_Number);  // "%lf" means double float

    // I have placed the Function call inside of the print statement.
```

```
    // This eliminates the need to create a variable to use as a "Temp Buffer"
    // to hold the return value before sending it to the print statement :)
    printf("Square root of %lf is %lf\n", User_Number, SquareRoot(User_Number));

    return 0;
    }

// ---> START Function Block

/*-Block-Comment-------------------------------------------------------------
Babylonian method to find the square root of a number.
SEE: Wikipedia - Methods of computing square roots

This method uses "approximation" to zone in on the result.
It starts with a Low approximation and a High approximation.
It slides both approximations until they (almost) meet at an approximation
of the square root to Precision decimal places.

Note: The calculation of Low and High may be infinite and never actually meet.
This is OK as we have enough accuracy for most purposes.
*/
double SquareRoot(double Radicand)
    {
    double High = Radicand;
    double Low = 1;
    double Precision = 0.000001;  // Decides the accuracy level (double float)

    while((High - Low) > Precision)  // Keep sliding until we reach "Precision".
        {  // v Get the average of Low + High for our new High Value.
        High = (Low + High)/2;
        Low = Radicand/High;  // Get the divisor for our new low value.
        }
    // Continue looping until Low and High are as close as "Precision" allows.

    return High;  // The value of High is our best approximation to return
    }

// END Function Block <---
```

FreeBASIC Example – Commented Source Code

Code: “Comments.bas”

```
'------------------------------------------------------------------------------
' Name:         FreeBASIC Comments example
' Purpose:      Illustrate Comments with a simple math Function
' Requires:
' Usage:        Read
'
'------------------------------------------------------------------------------
' Author:       Axle
' Copyright:    (c) Axle 2021
' Licence:      MIT No Attribution
' Created:      21/12/2021
' Modified:     19/02/2022
' Versioning    ("MAJOR.MINOR.PATCH") (Semantic Versioning 2.0.0)
' Script V:     0.0.2 (alpha)
'               (Alpha is functional, but missing features; Beta Is functional but
```

```
'               may still contain bugs; Release is fully functional and bug free)
' Encoding:     UTF-8
' Compiler V:   FreeBASIC Compiler TDM-GCC 9.2.0 32/64-bit
' OS Scope:     (Windows)
' UI Scope:     (CLI)
'-------------------------------------------------------------------------------
' NOTES:
'
'
'-------------------------------------------------------------------------------

' ---> START Library Imports
' END Library Imports <---
' ---> START Global Defines
Declare Function main_procedure() As Integer
Declare Function SquareRoot(Byval Radicand As Double) As Double
' END Global Defines <---

main_procedure()

Function main_procedure() As Integer
    Dim User_Number As Double

    Input "Enter a number to find the Square Root: ", User_Number

    Print "Square root of " & User_Number & " is " & SquareRoot(User_Number)

    Print "Press any key to continue..."
    Sleep  ' Sleep until a key is pressed
    Return 0
End Function

' ---> START Function Block

'-------------------------------------------------------------------------------
' Babylonian method to find the square root of a number.
' SEE: Wikipedia - Methods of computing square roots
'
' This method uses "approximation" to zone in on the result.
' It starts with a Low approximation and a High approximation.
' It slides both approximations until they (almost) meet at an approximation
' of the square root to Precision decimal places.
'
' Note: The calculation of Low and High may be infinite and never actually meet.
' This is OK as we have enough accuracy for most purposes.
'
Function SquareRoot(Byval Radicand As Double) As Double
    Dim High As Double = Radicand
    Dim Low As Double = 1.000000
    Dim Precision As Double = 0.000001  ' Decides the accuracy level (double
float)

    While((High - Low) > Precision)  ' Keep sliding until we reach "Precision".
        High = (Low + High)/2  ' Get the average of Low + High for our new High
Value.
        Low = Radicand/High  ' Get the divisor for our new low value.
    Wend  ' Continue looping until Low and High are as close as "Precision"
allows.
```

```
    Return High  ' The value of High is our best approximation to return
End Function

' END Function Block <---
```

Python 3 Example – Commented Source Code

Code: “Comments.py”

```
#-------------------------------------------------------------------------------
# Name:         Python 3 Comments example
# Purpose:      Illustrate Comments with a simple math Function
# Requires:
# Usage:        Read
#
#-------------------------------------------------------------------------------
# Author:       Axle
# Copyright:    (c) Axle 2021
# Licence:      MIT No Attribution
# Created:      21/12/2021
# Modified:     19/02/2022
# Versioning    ("MAJOR.MINOR.PATCH") (Semantic Versioning 2.0.0)
# Script V:     0.0.2 (alpha)
#               (Alpha is functional, but missing features; Beta Is functional but
#               may still contain bugs; Release is fully functional and bug free)
# Encoding:     utf-8
# Python V:     V3.9.x
# OS Scope:     See individual Modules (Windows, Unix, Python REPL)
# UI Scope:     See individual Modules (CLI)
#-------------------------------------------------------------------------------
# NOTES:
# Note: labels such as "var: float" or "() -> float" are Variable Type hints,
# and have no effect on the application.
#
#
#-------------------------------------------------------------------------------

## ---> START Library Imports
## END Library Imports <---
## ---> START Global Defines

def main():
    # ---> START Library Imports
    # END Library Imports <---
    User_Number: float

    print("Enter a number to find the Square Root: ", end='')
    User_Number = float(input())

    print("Square root of ", User_Number, " is ", SquareRoot(User_Number))

    input("press [Enter] to continue...")  # pause the program untill a key is
pressed

    return None

## ---> START Function Block
```

```
#-------------------------------------------------------------------------------
# Babylonian method to find the square root of a number.
# SEE: Wikipedia - Methods of computing square roots
#
# This method uses "approximation" to zone in on the result.
# It starts with a Low approximation and a High approximation.
# It slides both approximations until they (almost) meet at an approximation
# of the square root to Precision decimal places.
#
# Note: The calculation of Low and High may be infinite and never actually meet.
# This is OK as we have enough accuracy for most purposes.
def SquareRoot(Radicand: float) -> float:
    High: float = float(Radicand)
    Low: float = 1.000000
    Precision: float = 0.000001  # Decides the accuracy level (double float)

    while((High - Low) > Precision):  # Keep sliding until we reach "Precision".
        High = (Low + High)/2  # Get the average of Low + High for our new High
Value.
        Low = Radicand/High  # Get the divisor for our new low value.
        # Continue looping until Low and High are as close as "Precision" allows.

    return High  # The value of High is our best approximation to return

## END Function Block <---
## END Global Defines <---
if __name__ == '__main__':  # Program entry point
    main()

## Script Exit
exit()
```

Variables

Variables are a Named placeholder to store information

Variables are the information holding containers of an application. Like any storage box, the names can help us locate and use our data. Variable names are not there for the programming compiler or machine interpreter; they are there for the benefit of the programmer; the compiler will replace variable names with generated ID numbers. Using meaningful variable names will assist in tracing the data flow through your application during its initial creation as well as when we return to make changes to the code at a later date.

Variables hold the data that is being moved or manipulated within our application, and as such each variable contributes to the memory (RAM) that is required by our application on the host machine. It is important to limit our use of variables to the minimum that is required for the application to perform the required tasks. Duplicating the same data over multiple variables is a waste of the end user's RAM. In a large application this frequent duplication can amount to a significant and unnecessary use of memory.

Programming languages will make use of 2 types of memory storage; Static and Dynamic. Static memory is allocated alongside the application in RAM when it is executed and remains unchanged in size for the duration the application is running. Static memory allocates the maximum amount of memory that an application requires whether it is actually used or not. Dynamic memory is allocated

by the application at run time and is external from the application. Dynamic means that it can expand and use more memory or release and use less memory as required by the application. It is important that dynamic memory is released or “Freed '' when no longer required by the application. Not freeing dynamic memory can lead to excessive memory use as well as what is known as a memory leak where the dynamic variable is created repeatedly without being released after each use, resulting in a duplication of the same variable allocating a new block of RAM space alongside the previous allocations.

Another 2 terms used in variables and other data structures are “By Reference” and “By Value”. When we move data around inside of our application from one place to another and one variable or data structure to another we have 2 different ways in which we can do this. A variable name or data structure name is a label that points to where the contents of the envelope are stored in memory. When we create a new variable, often inside of a function we can point the new variable name to the same place in memory where the data exists for a previous variable instead of making a second copy of the data. We are using the new variable name to make a “Reference” of pointing to the data in memory from another variable. When we manipulate the data from a variable “By Reference” to another variable's data we also change the value of the data in the other variable. This is the default for C languages and brings in a somewhat confusing topic for some, called “Pointers”. Pointers are as the name says just a name/label that points to the location of the data in another variable.

BY Value means that when we make use of data from one variable to a new variable we make an exact copy of the data in the new variable. When we change the value of the new variable it has no effect on the original variable that we have copied it from. The variable name points to its own separate data in memory.

Variable names must not start with a digit or contain spaces 2_my_variable✗, my_variable_2✓, my variable✗ . Some special symbols such as underscore ‘_’ are allowed but underscore should be avoided as a prefix as they have special meaning in library code _My_Variable✗, My_Variable✓. Keywords must never be used as a variable name and some languages are case sensitive, meaning my_variable is different to My_Variable.

Although many languages and organisations will enforce a recommended naming convention for variables, in practice naming conventions are often intermixed. If in doubt, try to follow what is most common for the language and be consistent throughout your source code.

Letters vs Names: a = (b + c) / d; this is syntactically correct and makes the mathematical procedure more readable, but on the other hand, we really don’t know what is being calculated.
Average = (Total1 + Total2) / Sample_Number; tells us that we are getting an average from a sample of numbers, but can obscure the readability of the math equation. Sometimes we just have to roll with the context of the situation.

In the above letter example I would be inclined to comment above the equation if the variable has been declared in a position not readily visible.

Code:

```
// a =Average, b = Total1, c = Total2, d = Sample_Number
a = (b + c) / d;
```

Short meaningful variable names supported with brief comments makes for easy to read and follow code.

Case notation comes in a variety of styles and falls under the recommendations of Naming conventions above.

Multiple-word identifier formats	
Formatting	**Name(s)**
twowords	flatcase
TWOWORDS	UPPERFLATCASE
twoWords	(lower) camelCase, dromedaryCase
TwoWords	PascalCase, UpperCamelCase, StudlyCase
two_words	snake_case, pothole_case
TWO_WORDS	SCREAMING_SNAKE_CASE, MACRO_CASE, CONSTANT_CASE
two_Words	camel_Snake_Case
Two_Words	Pascal_Snake_Case
two-words	kebab-case, dash-case, lisp-case, spinal-case
TWO-WORDS	TRAIN-CASE, COBOL-CASE, SCREAMING-KEBAB-CASE
Two-Words	Train-Case, HTTP-Header-Case

Another common addition to this list is Hungarian notation which prefixes the variable name with a "Type" or the purpose.

Example:
iTotalScore (prefixed with 'I' for integer)
sPlayerNames (prefixed with 's' for string)

Personally, I use a lot of mixed notation in my code. Many frown upon it, but it makes my code more readable in most cases. Generally it is a mix of Hungarian, Pascal, Snake case, and Camel case depending upon the programming environment I am using.

Int_PayTotals, St_Name, **Str_Number, Totals_Pay**, **Totals_Expenses**.

Lower case and Snake case are the most common, such as **int_supplies**, **user_points**, and **total**.

The Windows Application Programming interface (API) uses Hungarian as well as Capitols for Type declarations, so it is common to encounter a mix of Snake case and Hungarian case in a Windows application written in C.

Examples: C vs Win-API

```
C:
Int my_number = 3
long my_number2 = 555454455445544554
float my_float = 3.14
char my_string[] = "A  string of characters"

Win-API
INT iMyNumber = 3
LONG lMyNumber = 555454455445544554
FLOAT fMyFloat = 3.14
CHAR sMyString[] = "A  string of characters"
```

Don't fret too much about getting the correct notations in the beginning as it really can be ambiguous even for the pro coder ☺ It's best to just focus on short meaningful words in a variable name.

Examples: Meaningful names

```
car_daily_expenses = 0
car_weekly_expenses = 0
car_monthly_expenses = 0
car_averag_weekly_expenses = 0
```

Variables are used within the "scope" of the application. The 2 most common uses of scope are "Global" meaning the variable can be used anywhere in the application and "Local" meaning the variable can only be used within a function or subroutine. Global variable names must be unique and should not be created elsewhere in the application as a local variable of the same name will be given precedence. Local variable names can be repeated across multiple functions as they are contained locally within that function and not used.

In C and FreeBASIC there is another keyword called "extern". This is used in library files that are linked from the main source file. "extern" declares that a variable is "global" to all files that have included this file. "extern" is used in a header file "foo.h" which declares and sets the definitions of any variable used in its associated source file "foo.c". I won't be showing its use here but will cover it in another section under "Libraries".

Another type of named data holder is called a Constant or Literal. Constants and Literals are immutable meaning they can't be changed or deleted, whereas Variables are mutable meaning they can be changed or deleted. Constants and literals remain unchanged during the entire execution of the application. In most cases we will use what is called a "String literal", whereas Constants are generally related to numeric values such as Integers and Floating Point. Constants are most commonly used in a global configuration context.

Additional information:

https://en.wikibooks.org/wiki/C_Programming/Variables

Examples: Global, Local, Variable and Constant.

```
Rem Global scope
    my_variable
    Global my_variable
    Constant my_constant = 9
    my_str_literale = "A string literal"

my_function()
Rem Local scope
    my_variable
    Constant my_constant = 8
    My_str_literal = "A string literal too"
```

Variables can be made up of single data types in some languages and are referred to as "Typed" languages. The most common "Types"used are Boolean, Integer, float, and String although most languages will make use of many different data types.

Some scripting languages will allow any data type in any given variable and are referred to as "Loosely Typed" and "Typeless" languages. The language interpreter will work out which is the most suitable type at runtime based upon the context in which it is used.

Variable declaration, initialization and assignment. When a variable is first created it is said to have been "Declared". Declaration can give a data type and declare that the variable name exists but doesn't give a value for the variable to hold **int var**. Initialization is the first time of assigning a value to the variable name although we can initialize a variable with 0 or null without actually assigning it a value **int var = None**. Assignment is any occasion in which the variable is given a value to hold **var = 5**.

We can and often do declare, initialize and assign a value in the same step **int var = 5**.

Examples: Variable Data Type declaration + initialization and assignment

```
Rem Typed Language
BOOL truth_test;  // declaration
truth_test = 1;  // True (truth_test = 0; // False)  // initialization and
assignment
INT my_integer;   // declaration
my_integer = 22;  // initialization and assignment
FLOAT my_floating_point;   // declaration
my_floating_point = 1.41421  // initialization and assignment
STRING my_string_of_characters;  // declaration
my_string_of_characters = "-C-h-a-r-a-c-t-e-r-s- -o-n- -a- -s-t-r-i-n-g-" //
initialization and assignment

Rem Loosely Typed Language
truth_test;  // declaration
truth_test = 1;  // True (truth_test = 0; // False)  // initialization and
assignment
my_integer;  // declaration
my_integer = 22;  // initialization and assignment
my_floating_point;  // declaration
my_floating_point = 1.41421 // initialization and assignment
my_string_of_characters;  // declaration
my_string_of_characters = "-C-h-a-r-a-c-t-e-r-s- -o-n- -a- -s-t-r-i-n-g-" //
initialization and assignment
```

Language: C

Code: "Variables.c"

```
//------------------------------------------------------------------------------
// Name:        Variables.c
// Purpose:     Example
//
// Platform:    Win64, Ubuntu64
//
// Author:      Axle
// Created:     31/01/2022
// Updated:     19/02/2022
// Copyright:   (c) Axle 2022
// Licence:     MIT No Attribution
//------------------------------------------------------------------------------

#include <stdio.h>  // include standard library headers
#include <stdlib.h>
#include <string.h>

// ---> declare functions
```

```
void test_const(void);
void test_local_vs_global(void);
void test_variable(void);
void win_api_test(void);
// ---> MACROS
#define MAXVALUE 128

// ---> Global declare & defines
const int config_max_str_len = 32;
char *my_lstring = "Global string";  // *pointer to a literal.
//char my_lstring[] = "Global string";  // same declaration as previous.
int my_variable_integer = 6;  // an Integer variable
char my_variable_string[64] = {'\0'};  // a String variable.


int main(int argc, char *argv[])  // Main procedure
    {
    // ---> Local declare & defines
    char *my_lstring = "Local string";  // *pointer to a literal.
    //char my_lstring[] = "Global string";  // same declaration as previous.
    int my_variable_integer = 3;  // an Integer variable
    char my_variable_string[64] = {'\0'};  // a String variable.
    char local_lstring[] = "Local to main()";

    // using a simple MACRO
    printf("The Maximum value allowed = %d\n\n", MAXVALUE);

    // ---> Tests
    // config_max_str_len += 1;  // [Error] assignment of read-only variable
'config_max_str_len'
    printf("// Test our Global const in local scope\n");
    printf("main(), config_max_str_len = %d\n", config_max_str_len);  // cannot be
altered
    test_const();  // test in a different local scope.
    printf("\n");

    // The Local overrides global variables!
    // Although they have the same name they are different variables.
    // Global variables should be used with great care and be made up of unique
names.
    printf("// The Local literal variable declaration overrides the Global
variable.\n");
    printf("main(), my_lstring = %s\n", my_lstring);  // Local overrides global!
    printf("main(), my_variable_integer = %d\n", my_variable_integer);  // Local
overrides global!
    printf("main(), local_lstring = %s\n", local_lstring);
    test_local_vs_global();  // test in a different local scope.
    printf("\n");

    // The Local overrides global variables!
    printf("// The literal is copied to the local variable in main()...\n");
    strcpy(my_variable_string, "my_variable_string Local to main()");
    printf("main(), my_variable_string = %s\n", my_variable_string);
    test_variable();// test in a different local scope.
    // After changing the value of the Global variable...
    printf("// The variable Local to main() has not altered.\n");
    printf("main(), my_variable_string = %s\n", my_variable_string);
    printf("\n");
```

```
    printf("Press the [Enter] key to continue...");
    getchar();  // Pause the program until a key is pressed
    return 0;
    }

void test_const(void)  // This is a void function (aka Subroutine).
    {
    // cannot be altered
    printf("test_const(), config_max_str_len = %d\n", config_max_str_len);
    }

void test_local_vs_global(void)  // This is a void function (aka Subroutine).
    {
    char local_lstring[] = "Local to test_local_vs_global()\n";
    // We have not blocked the global definition with the same local name.
    printf("// and to the Global variable in test_variable().\n");
    printf("test_lstring(), my_lstring = %s\n", my_lstring);
    printf("main(), my_variable_integer = %d\n", my_variable_integer);
    printf("main(), local_lstring = %s\n", local_lstring);
    }

void test_variable(void)  // This is a void function (aka Subroutine).
    {
    printf("test_variable(), my_variable_string = %s\n", my_variable_string);
    // The following copies the "string literal" into the Global variable.
    // The "my_variable_string Global" in the following function is a
    // true string literal as it has no variable associated with it until it
    // is copied into my_variable_string.
    strcpy(my_variable_string, "my_variable_string Global");
    printf("test_variable(), my_variable_string = %s\n", my_variable_string);
    }
```

Language: FreeBASIC

Code: “Variables.bas”

```
'--------------------------------------------------------------------------------
' Name:         Variables.bas
' Purpose:
'
' Platform:     Win64, Ubuntu64
'
' Author:       Axle
' Created:      31/01/2022
' Updated:      19/02/2022
' Copyright:    (c) Axle 2022
' Licence:      MIT No Attribution
'--------------------------------------------------------------------------------

' ---> declare functions
Declare Function main_procedure() As Integer
Declare Sub test_const()
Declare Sub test_local_vs_global()
Declare Sub test_variable()

' ---> MACROS
#define MAXVALUE 128
```

```
' ---> Global declare & defines
Const As Integer config_max_str_len = 32
Dim Shared As String my_lstring
my_lstring = "Global string"  ' Note: String is a keyword and can't be used as a
variable.
Dim Shared As Integer my_variable_integer = 6  ' an Integer variable
Dim Shared my_variable_string As String  ' a String variable.

main_procedure()

Function main_procedure() As Integer  ' Main procedure

    ' ---> Local declare & defines
    Dim As String my_lstring = "Local string"  ' pointer to a literal.
    Dim As Integer  my_variable_integer = 3  ' an Integer variable
    Dim my_variable_string As String  ' a String variable.
    Dim As String local_lstring = "Local to main_procedure()"

    ' using a simple MACRO
    Print "The Maximum value allowed = "; MAXVALUE
    Print ""

    ' ---> Tests
    ' config_max_str_len += 1 ' error 119: Cannot modify a constant, before '+'
    Print "' Test our Global const in local scope"
    Print "main_procedure(), config_max_str_len = "; config_max_str_len  ' cannot
be altered
    test_const()  ' test in a different local scope.
    Print ""

    ' The Local overrides global variables!
    ' Although they have the same name they are different variables.
    ' Global variables should be used with great care and be made up of unique
names.
    Print "' The Local literal variable declaration overrides the Global
variable."
    Print "main_procedure(), my_lstring = "; my_lstring  ' Local overrides global!
    Print "main_procedure(), my_variable_integer = "; my_variable_integer  ' Local
overrides global!
    Print "main_procedure(), local_lstring = "; local_lstring
    test_local_vs_global()  ' test in a different local scope.
    Print ""

    ' The Local overrides global variables!
    Print "' The literal is copied to the local variable in main()..."
    my_variable_string = "my_variable_string Local to main()"
    Print "main_procedure(), my_variable_string = "; my_variable_string
    test_variable()  ' test in a different local scope.
    ' After changing the value of the Global variable...
    Print "' The variable Local to main() has not altered."
    Print "main_procedure(), my_variable_string = "; my_variable_string
    Print ""

    Print "Press any key to continue..."
    Sleep  ' Sleep until a key is pressed
    Return 0
End Function

Sub test_const()
```

```
    ' cannot be altered
    Print "test_const(), config_max_str_len = "; config_max_str_len
End Sub

Sub test_local_vs_global()
    Dim As String local_lstring = "Local to test_local_vs_global()"
    ' We have not blocked the global definition with the same local name.
    Print "' and to the Global variable in test_variable()."
    Print "test_lstring(), my_lstring = "; my_lstring
    Print "main_procedure(), my_variable_integer = "; my_variable_integer
    Print "main_procedure(), local_lstring = "; local_lstring
End Sub

Sub test_variable()
    Print "test_variable(), my_variable_string = "; my_variable_string
    '  the following copies the "string literal" into the Global variable.
    ' the "my_variable_string Global" in the following function is a
    ' true string literal as it has no variable associated with it until it
    ' is copied into my_variable_string.
    my_variable_string = "my_variable_string Global"
    Print "test_variable(), my_variable_string = "; my_variable_string
End Sub
```

Language: Python 3

Code: “Variables.py”

```
#-------------------------------------------------------------------------------
# Name:        Variables.py
# Purpose:
#
# Platform:    REPL, Win64, Ubuntu64
#
# Author:      Axle
# Created:     31/01/2022
# Updated:     19/02/2022
# Copyright:   (c) Axle 2022
# Licence:     MIT No Attribution
#-------------------------------------------------------------------------------

## ---> MACROS
MAXVALUE = 128

## ---> Global declare & defines
config_max_str_len = 32  # An Integer
my_lstring = "Global string"  #  Pointer to a String literal.
my_variable_integer = 6  # An Integer
my_variable_string =""  # A String variable.

def main():

    # ---> Local declare & defines
    my_lstring = "Local string"  # *pointer to a literal.
    my_variable_integer = 3  # an Integer variable
    my_variable_string = ""  # a String variable.
    local_lstring = "Local to main()"

    # Using a simple MACRO
```

```
    print("The Maximum value allowed = ", MAXVALUE)

    # ---> Tests
    # config_max_str_len += 1;  // [Error] assignment of read-only variable
'config_max_str_len'
    print("# Test our Global const in local scope")
    print("main(), config_max_str_len = ", config_max_str_len)  # cannot be
altered
    test_const()  # test in a different local scope.
    print("")

    # The Local overrides global variables!
    # Although they have the same name they are different variables.
    # Global variables should be used with great care and be made up of unique
names.
    print("# The Local literal variable declaration overrides the Global
variable.")
    print("main(), my_lstring = ", my_lstring)  # Local overrides global!
    print("main(), my_variable_integer = ", my_variable_integer)  # Local
overrides global!
    print("main(), local_lstring = ", local_lstring)
    test_local_vs_global()  # test in a different local scope.
    print("")

    # The Local overrides global variables!
    print("# The literal is copied to the local variable in main()...")
    my_variable_string = "my_variable_string Local to main()"
    print("main(), my_variable_string = ", my_variable_string)
    test_variable()  # Test in a different local scope.
    # After changing the value of the Global variable...
    print("# The variable Local to main() has not altered.")
    print("main(), my_variable_string = ", my_variable_string)
    print("");

    input("Press [Enter] to exit.")  # Wait for keypress to keep the console open.
    return None
## ---> END of MAIN MENU <---

def test_const():
    # Cannot be altered
    print("test_const(), config_max_str_len = ", config_max_str_len)

def test_local_vs_global():
    local_lstring = "Local to test_local_vs_global()"
    global my_lstring
    global my_variable_integer
    # We have not blocked the global definition with the same local name.
    print("# and to the Global variable in test_variable().")
    print("test_lstring(), my_lstring = ", my_lstring)
    print("main(), my_variable_integer = ", my_variable_integer)
    print("main(), local_lstring = ", local_lstring)

def test_variable():
    global my_variable_string
    print("test_variable(), my_variable_string = ", my_variable_string)
    # The following copies the "string literal" into the Global variable.
    # The "my_variable_string Global" in the following function is a
    # true string literal as it has no variable associated with it until it
    # is copied into my_variable_string.
```

```
    my_variable_string = "my_variable_string Global"
    print("test_variable(), my_variable_string = ", my_variable_string)


if __name__ == '__main__':
    main()
```

Arrays

Arrays are a special type of variable or really a collection of variables. An array usually contains only a single data type for its variable and is written in the form of **Variable_Name[Value_index_number]** where the index number points to the numbered data slots in the variable.

It is useful to think of the variable slots (elements) the same as we would view a table or excel spreadsheet with Rows and Columns so a single dimension array array[5] is the same as a single Row in a table with 5 Columns.

In a 2 dimensional array **Array_Name[Row_index_number][Col_index_number]** the first dimension points to the Row number Array _Name[**Row_index_number**][Col_index_number] and the second dimension points to the Column number and it's value Array_Name[Row_index_number][**Col_index_number**].

It's important to note that the most right dimension in an array is the index that holds the actual values, so **Array _Name[index_number = Value]** and **Array_Name[index_number][index_number = Value]** as well as a 3D array **Array_Name[index_number][index_number][index_number = Value]**

In reality there is an extra dimension attached to the right hand index that holds the actual data but it is generally hidden from view when adding or reading the data from the array. In my C examples you will see the extra dimension when creating String arrays as a string is really just a 1D array of characters.

A String is created as **array[number of characters(aka Value)]** and accessed as **Print("array")**, a 1D string array is created as **array[Columns][Value]** and the hidden value is accessed as **Print(array[Column])**. A 2D array of **array[Rows][Columns][Value]** is accessed as **array[Rows][Columns]**, so the Value section is hidden in the background **array[Rows][Columns]***[Value].*

Arrays are one of the most used and important data structures used in any computer program. The 2 most common arrays that you will encounter are single and 2 dimensional arrays. In effect every time you use a string of characters this is a single dimension array. A string would be represented as **MyString[64]** with a maximum number of characters in the string to be less than 64. High level languages will often use a static allocation of the maximum amount of characters, usually **[64,000]** and the array allocation is not required so we can simply create a string variable **MyString** without having to allocate space for the data.

A data array such as we use in Access or a CSV data file has 2 dimensions of Row and Column. It will always be a 2 dimensional array with rows being the first index in the array **array[Row][Col]** "Row-wise".

When we think about the graphics output on our monitor this is a 1 dimensional array of z which contains 2 dimensional arrays of array[x][y] coordinates. This can be shown as **array[z][array[x][y]]**. This is also a special type of array known as a "Data Structure" or a "Structure of arrays".

In languages such as Python, Linked Lists and Dictionaries are a structure with many indexed arrays within it.

Example C data structure

```
struct Books { //The data structure
   char  title[50]; //The arrays
   char  author[50]; //The arrays
   char  subject[100]; //The arrays
   int   book_id; //The arrays
} book;
```

Matrices are a special type of array that only have 2 dimensions **array[x][y]**, whereas arrays in general can have any dimension from 1 to ∞. xy arrays read the rows and columns with a column-wise orientation **array[x_Col][y_Row]** as opposed to a table which reads Row-wise **array[y_Row][x_Col]**.

Vectors are special single dimensional arrays known as a "Data Structure" with value pairs **array[index] = (x,y)** and can also be used in vector algebra.

Remember when we think of a 1 or 2 dimensional array such as **Address[street_number][Name]** the actual data is in a hidden slot on the right side of the array index **Address[street_number][Name]**[Name = Data]. In effect this is like an index (pointer) that points to the memory location of the actual data.

Language: C

Code: "Arrays.c"

```
//------------------------------------------------------------------------------
// Name:        Arrays
// Purpose:     Example
//
// Platform:    Win64, Ubuntu64
//
// Author:      Axle
// Created:     03/02/2022
// Updated:     19/02/2022
// Copyright:   (c) Axle 2022
// Licence:     MIT No Attribution
//------------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// C preprocessor replaces all occurrence in the source of ROW with 5 etc.
#define ROWS 5
#define COLUMNS 5
#define STRMAX 32  // A buffer of 32 chars to hold our strings.

int main()  // Main procedure
```

```
    {
    const int Rows = ROWS;
    const int Columns = COLUMNS;
    char TempBuffer[STRMAX] = {'\0'};  // array of 32 characters, aka String.

    // defining a 2D array of array[Rows][Columns], initialised to empty.
    static char Table[ROWS][COLUMNS][STRMAX] = {'\0'};  // initialised to null
    // static char canvas[5][5][32] = {'\0'};  // Actual values of above.

    int Row_y;  // Counter for rows.
    int Col_x;  // Counter for columns.
    int Offset = 1;  // Origin zero offset (0|1)
    // Fill the array with some data.
    for(Row_y = 0; Row_y < Rows; Row_y++)  // Count through each row.
        {
        for(Col_x = 0; Col_x < Columns; Col_x++)  // Count through each column.
            {
            // Build a string with values for each cell. The offset starts
            // the Row and columns count at 1 instead of 0.
            sprintf(TempBuffer, "[R:%d,C:%d]", Row_y + Offset, Col_x + Offset);
            // Copy the string to the cell position.
            strcpy(Table[Row_y][Col_x], TempBuffer);
            }
        }

    // Print the array to the console.
    for(Row_y = 0; Row_y < Rows; Row_y++)  // Count through each row.
        {
        for(Col_x = 0; Col_x < Columns; Col_x++)  // Count through each column.
            {
            printf("%s ", Table[Row_y][Col_x]);  // No line breaks.
            }
        printf("\n");  // Print a line break to start next row.
        }

    printf("Press the [Enter] key to continue...");
    getchar();  // Pause the program until a key is pressed
    return 0;
    }
```

Language: FreeBASIC

Code: “Arrays.bas”

```
'--------------------------------------------------------------------------------
' Name:         Arrays
' Purpose:      Example
'
' Platform:     Win64, Ubuntu64
'
' Author:       Axle
' Created:      03/02/2022
' Updated:      19/02/2022
' Copyright:    (c) Axle 2022
' Licence:      MIT No Attribution
'--------------------------------------------------------------------------------

Const ROW = 5
```

```
Const COLUMN = 5

Declare Function main_procedure() As Integer

main_procedure()

Function main_procedure() As Integer  ' ---> Main procedure

    Dim As Integer Rows = ROW
    Dim As Integer Columns = COLUMN
    Dim As String TempBuffer = ""  ' array of characters, aka String.

    ' defining a 2D array of array[Rows][Columns], initialized to empty.
    Dim As String Table(ROW, COLUMN)  ' initialized to null
    ' static char Table(5, 5)  '  Actual values of above.

    Dim As Integer Row_y  ' Counter for rows.
    Dim As Integer Col_x  ' Counter for columns.
    Dim As Integer Offset = 1  '  Origin zero offset (0|1)
    ' Fill the array with some data.
    For Row_y = 0 To Rows -1 Step 1  ' Count through each row.
        For Col_x = 0 To Columns -1 Step 1  ' Count through each column.
            ' Build a string with values for each cell. The offset starts
            ' the Row and columns count at 1 instead of 0.
            TempBuffer = "[R:" & (Row_y + Offset) & ",C:" & (Col_x + Offset) & "]
"
            ' Copy the string to the cell position.
            Table(Row_y, Col_x) = TempBuffer
        Next Col_x
    Next Row_y

    ' Print the array to the console.
    For Row_y = 0 To Rows -1 Step 1  ' Count through each row.
        For Col_x = 0 To Columns -1 Step 1  ' Count through each column.
            Print Table(Row_y, Col_x);  ' No line breaks.
        Next Col_x
        Print ""  ' Print a line break to start next row.
    Next Row_y

    Print "Press any key to continue..."
    Sleep  ' Sleep until a key is pressed
    Return 0
End Function  ' END main_procedure <---
```

Language: Python 3

Code: “Arrays.py”

```
#--------------------------------------------------------------------------------
# Name:         Arrays
# Purpose:      Example
#
# Platform:     REPL, Win64, Ubuntu64
#
# Author:       Axle
# Created:      03/02/2022
# Updated:      19/02/2022
# Copyright:    (c) Axle 2022
```

```
# Licence:     MIT No Attribution
#-------------------------------------------------------------------------------

ROW = 5
COLUMN = 6

def main():

    Rows = ROW
    Columns = COLUMN
    TempBuffer = ""  # array of characters, aka String.

    # defining a 2D array of array[Rows][Columns], initialized to empty.
    Table = [[None]* COLUMN for _ in range(ROW)]  # initialized to null
    # static char Table(5, 6)  # Actual values of above.

    Row_y = 0  # Counter for rows.
    Col_x = 0  # Counter for columns.
    Offset = 1  # Origin zero offset (0|1)
    # Fill the array with some data.
    for Row_y in range(0,  Rows):  # Count through each row.
        for Col_x in range(0,  Columns):  # Count through each column.
            # Build a string with values for each cell. The offset starts
            # the Row and columns count at 1 instead of 0.
            TempBuffer = "[R:" + str(Row_y + Offset) + ",C:" + str(Col_x + Offset)
+ "] "
            # Copy the string to the cell position.
            Table[Row_y][Col_x] = TempBuffer

    # Print the array to the console.
    for Row_y in range(0,  Rows):  # Count through each row.
        for Col_x in range(0,  Columns):  # Count through each column.
            print(Table[Row_y][Col_x], end='')  # No line breaks.
        print("")  # Print a line break to start next row.

    input("Press [Enter] to exit.")
    return None
# END main <---

if __name__ == '__main__':
    main()
```

Data (Variable) Types

Most data types are built from primitive integers and are dependent upon the machine in which they are used. Pointers are a special data type that points to an *address range* in memory containing an integer type.

Pointer to an integer in memory												
Address	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011
Content					120					0004		
Variable					a							

Pointer										P*		

```
Variable a holds an integer value of Dec 120 at memory address 0004.
Pointer p* holds a Hex value that is the memory location of a.
It is said that p* “points” to the memory location of a.
```

Integers can have different widths depending upon the type of data they need to be able to hold. For example and ASCII/ANSI char is 1 byte or 8 bits wide (256 possible values) also known as int-8, a UTF-16 character is 2 bytes or 16 bits wide (65,536 possible values) and an int integer is 4 bytes or 32 bits wide (4,294,967,296 possible values). These data types generally do not change between 16 bit, 32 bit and 64 bit operating systems, but may differ between Windows and UNIX. Windows 64 bit uses a lot of 32 bit code in its C runtime libraries and may present anomalies with warnings for some printf type formatters in C.

The CPU architecture actually states the maximum value that a *pointer can hold, so a 16 bit machine can only hold a maximum address range of 65,536 possible values aka 64KiB of memory, and a 32 bit machine has a maximum addressable memory range of 3.99GiB which is why 32 bit Operating systems could only use 4GiB sticks of RAM.

Pointers “Point” to the starting address of our data (The contents of a variable) and a variable holds the contents within the width of a data type.

Below is a table with some common type widths.

Integer Data Types		
Type Name	**32-bit Size**	**64-bit Size**
char	1 byte	1 byte
short	2 bytes	2 bytes
int	4 bytes	4 bytes
long	4 bytes	8 bytes
long long	8 bytes	8 bytes
Size_t	*Pointer width (4 bytes)	*Pointer width (8 bytes)
Floating-Point Data Types		
Type Name	**32-bit Size**	**64-bit Size**
float	4 bytes	4 bytes
double	8 bytes	8 bytes
long double	16 bytes	16 bytes
Unicode Character Types		
Type Name	**32-bit Size**	**64-bit Size**
wchar_t | UCS_2	2 bytes	2 bytes
char8_t | (UTF-8)	1 byte(1 to 4 x 1 byte)	1 byte (1 to 4 x 1 byte)
char16_t | UTF-16	2 bytes	2 bytes
char32_t | UTF-32	4 bytes	4 bytes

It is perfectly acceptable to encode Multibyte UTF-8 in unsigned int, unsigned char, byte, or char8_t as they are all derivatives of unsigned int.
The same with char16_t would be the same as unsigned short.

Most data types described in different programming languages are just the above primitive integers that have been renamed.

Characters.

At the most basic level, characters are made up from the ASCII character set. A single byte (8 bits) contains 256 possible values. The first 7 bits 0 to 127 make up the ASCII characters set which include the print control characters from 0 to 31 and the print or keyboard characters from 32 to 127. Depending upon the "Type" selected the first 7 bits 0 to 127 will relate as an integer value or as a char value. The 8th bit is what is known as the significant bit and will determine if the last 128 values of the byte are used. An integer can use from 0 to 127 as positive values, and will become the equivalent 128 negative values when bit 8 is turned on. This leads to a physical property where a "signed"byte can contain 2 zeros. -0 and +0 and is known as the 2s complement. Most languages programmatically work around it so you don't have to be concerned about it.

For characters, the second 128 values from 128 to 255 make up the extended character set and are determined by the "code page" or language for the region in which the computer is used. Most English computers will use IBM Code Page 437 or Microsoft CP-1252.

When the full numeric set of a byte from 0 to 255 is used it is often referred to as a binary as opposed to an ASCII or text file.

UTF-8 is a variable byte character encoding that can be made of 1 to 4 bytes. The number of bytes can vary for each character in a string of text. The first 7 bits of the first byte in UTF-8 relate directly to the values used in ASCII and are interchangeable. So ASCII converts directly to UTF-8.

C does not have primitive definitions for Unicode in the standard library and a third party UTF-8/16 etc library is required to handle Unicode characters. The MSVCRT does have support for wchar_t or UCS-2. FreeBASIC is the same as C and a 3rd party library is required for Unicode. Python 3 uses UTF-8 encoding by default, but will require additional libraries for other encodings.

Also note that printing/displaying UTF characters is also dependent upon the text abilities of the display interface such as the OS console/terminal emulator or browser.

Although it is claimed that Windows uses UTF-16 since W2000 it still defaults to UCS-2 which is not fully compatible with UTF-16. So when encoding in UTF it is important to note the difference. In C ANSI is type char, Win-API CHAR; UCS-2 in C is type wchar_t, Win-API WCHAR (2 byte Fixed width); UTF-16 in C is utf16_t and are Win-API implementation defined.

In C and FreeBASIC Char = 'C' = Byte = int 67 = ASCII Dec 67 = ASCII Hex 43 = 0x43 = &h43 = Asc("C") = Chr(67). These are all types that describe the character 'C'.

Language: C

Code: "Types.c"

```
//------------------------------------------------------------------------------
// Name:        Types.c
// Purpose:     Examples
//
// Platform:    Win64, Ubuntu64
//
// Author:      Axle
// Created:     07/03/2022
// Updated:
// Copyright:   (c) Axle 2022
// Licence:     MIT No Attribution
```

```c
//------------------------------------------------------------------------------

// Required to to enable Long Long in MinGW and MSVCRT.dll
//
#ifdef _WIN32
#   ifdef _WIN64
#       define __USE_MINGW_ANSI_STDIO 1
#   endif
#endif

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

int main(int argc, char *argv[])  // Main procedure
{
    int a = 16;
    // sizeof() returns the width of the type in bytes.
    // sizeof() operator will return a type size_t. size_t will always be the
    // width of the pointer type for the system where it is used.
    // 16 bit, 32 bit, 64 bit. As I am using 64 bit in the example It is 8 bytes
    // wide unsigned long long.
    // long long has a different byte width between Windows x64 and Linux x64
    // So I am using the %zu modifier instead of %llu
    printf("Size of variable a : %zu\n", sizeof(a));
    printf("Size of int data type : %zu\n", sizeof(int));
    printf("Size of char data type : %zu\n", sizeof(char));
    printf("Size of float data type : %zu\n", sizeof(float));
    printf("Size of double data type : %zu\n", sizeof(double));
    printf("Size of short data type : %zu\n", sizeof(short));
    printf("Size of long data type : %zu\n", sizeof(long));
    printf("Size of long long data type : %zu\n", sizeof(long long));
    printf("Size of size_t data type : %zu\n", sizeof(size_t));
    getchar();
    return 0;
}
```

Language: FreeBASIC

Code: “Types.bas”

```
'------------------------------------------------------------------------------
' Name:         Types.bas
' Purpose:      Examples
'
' Platform:     Win64, Ubuntu64
'
' Author:       Axle
' Created:      07/03/2022
' Updated:
' Copyright:    (c) Axle 2022
' Licence:      MIT No Attribution
'------------------------------------------------------------------------------

Declare Function main_procedure() As Integer

main_procedure()

Function main_procedure() As Integer  ' Main procedure
```

```
        Dim As Integer a = 16
        ' SizeOf() operator will return the byte width of the type.
        Print "Size of variable a : "; Sizeof(a)
        Print "Size of Integer data type : "; Sizeof(Integer)
        Print "Size of String data type : "; Sizeof(String)
        Print "Size of Single(float) data type : "; Sizeof(Single)
        Print "Size of Double data type : "; Sizeof(Double)
        Print "Size of Short data type : "; Sizeof(Short)
        Print "Size of Long data type : "; Sizeof(Long)
        Print "Size of longInt data type : "; Sizeof(Longint)

    Getkey
    Return 0
End Function  ' END main_procedure <---
```

Language: C

Code: "Types.py"

```
#-------------------------------------------------------------------------------
# Name:        Functions.py
# Purpose:     Examples
#
# Platform:    REPL*, Win64, Ubuntu64
#
# Author:      Axle
# Created:     22/02/2022
# Updated:
# Copyright:   (c) Axle 2022
# Licence:     MIT No Attribution
#-------------------------------------------------------------------------------

import sys

def main():


    # Unfortunately python does not contain any primitive data types. The best
    # we can achieve is the total size of the object but we cant obtain the
    # size of the primitives in the object.
    #    struct _longobject {
    #    long ob_refcnt;
    #    PyTypeObject *ob_type;
    #    size_t ob_size;
    #    long ob_digit[1];
    #};
    a = 16
    print("Size of variable a : ", sys.getsizeof(a))
    print("Size of Integer data type : ", sys.getsizeof(int))
    print("Size of String data type : ", sys.getsizeof(str()))
    print("Size of Double data type : ", sys.getsizeof(float()))

    input("")
if __name__ == '__main__':
    main()
```

Statements, expressions and Procedures

The core commands that give instructions of what to do as well as calculations. Often referring to a line or group of commands.

When you first begin coding you will be surrounded by a whirlpool of terminology that can feel a little like alphabet soup. Don't be too overwhelmed by it all at the start. It is a knowledge that comes with time and experience in the programming environment. Even experienced programmers can find terminology a little difficult as many terms will blend together in practice.

Expressions relate to any code that returns a resulting value. So **Value = (9 /3)** is an expression, whereas Statements perform some form of action. So **print("My Value = ", Value)** is a statement.

While, for loops are statements that contain an expression. So **For I = 0 To 9; Next I** has a "For Next statement with an internal expression "I = 0 to 9" where Next I is the resulting value.

Procedures tend to describe the steps that arrive at a destination and are made up of many statements and expressions, and a Routine is a group of expressions, statements and procedures that is repeatable.

Throughout this booklet the focus has been on what is known as Imperative Procedural programming. Procedural programming is a type of Imperative programming that focuses on the detailed steps to achieve an action or result.

Another type of programming paradigm is known as Declarative programming. Declarative programming is only interested in the result, and pays little attention as to how the result is achieved.

Low abstraction level programming such as the C language is most often used in a Procedural orientation, whereas some high abstraction level languages such as SQL can have a Declarative orientation. Object Oriented programming is a higher abstraction layer built on top of the Procedural paradigm, so OOP programming is fundamentally procedural, even though it may feel Declarative in some usage scenarios.

It is not uncommon for programming languages to have the ability to make use of a number of different orientations or programming paradigms.

Wikipedia - Comparison of programming paradigms

Language: C

Code: "Statements.c"

```
//------------------------------------------------------------------------
// Name:        Statements(ST), expressions(EX) and Procedures(PR)
// Purpose:     Example
//
// Platform:    Win64, Ubuntu64
//
// Author:      Axle
// Created:     31/01/2022
// Updated:     19/02/2022
// Copyright:   (c) Axle 2022
// Licence:     MIT No Attribution
```

```
//--------------------------------------------------------------------------

#include <stdio.h>  // include standard library headers
#include <stdlib.h>
#include <string.h>

// You can change the radius and pen size below.
#define RADIUS 10  // Do NOT exceed 20! 80 char MAX Console width.
#define PEN 10  // About 1:1 with the Radius to get single char.

int main()  // PR Main procedure
    {
    // Defining a 2D array of array[Rows][Columns], initialised to empty.
    static char canvas[RADIUS * 5][RADIUS * 5][256] = {'\0'};
    // Changing the following String characters will change the circle display.
    char Foreground[] = "O ";  // EX add space after chr to make circle (MAX 2
chrs)
    char Background[] = ". ";  // EX add space after chr to make circle (MAX 2
chrs)

    const int Radius = RADIUS;  // EX
    const int Tolerance = PEN;  // EX Larger numbers will create a wider drawing
pen.

    int Row_yy = 0;  // EX counters for array position.
    int Col_xx = 0;  // EX counters for array position.
    int Row_y;  // ST Counter range of circumference. -Radius to +radius.
    int Col_x;  // ST Counter range of circumference. -Radius to +radius.
    for(Row_y = -Radius; Row_y <= Radius; Row_y++)  // PR + EX
        {
        Col_xx =0;  // EX reset the column counter for each row.
        for(Col_x = -Radius; Col_x <= Radius; Col_x++)  // PR + EX
            {
            // Test if it is at the radius
            int equation = Row_y*Row_y + Col_x*Col_x - Radius*Radius;  // EX
            if (abs(equation) < Tolerance)//ST
                {
                strncpy(canvas[Row_yy][Col_xx], Foreground, 2);  //(a) ST
                //printf("%s", Foreground);//(b) ST
                }
            else
                {
                strncpy(canvas[Row_yy][Col_xx], Background, 2);  //(a) ST
                //printf("%s", Background);  //(b) ST
                }
            //printf("%s", canvas[Row_yy][Col_xx]);  //(c) ST
            Col_xx += 1;  // EX Increment Columns

            }
        Row_yy += 1;  // EX Increment Rows
        //printf("\n");  // ST
        }  // END PR

    // Commenting out the loop below, and then enabling the
    // printf statements above will print the circle directly
    // rather than populating the array first and printing later :)
    // Comment out (b) and enable (c) to print directly from the array/list.
    // Or comment out (a)(c) and enable (b) to print directly without the
array/list.
```

```
    // Print the rows and columns from our canvas array.
    // The last increment of Row_yy and Col_xx from the loop above
    // contains the correct lengths for the following loop.
    for(Row_y = 0; Row_y < Row_yy; Row_y++)// PR + EX
        {
        for(Col_x = 0; Col_x < Col_xx; Col_x++)  // PR + EX
            {
            printf("%s", canvas[Row_y][Col_x]);  // ST
            }
        printf("\n");  // ST
        }  // END PR

    printf("Press the [Enter] key to continue...");
    getchar();  // Pause the program until a key is pressed
    return 0;  // EX
    }  // END PR
```

Language: FreeBASIC

Code: “Statements.bas”

```
'--------------------------------------------------------------------------------
' Name:         Statements(ST), expressions(EX) and Procedures(PR)
' Purpose:      Example
'
' Platform:     Win64, Ubuntu64
'
' Author:       Axle
' Created:      31/01/2022
' Updated:      19/02/2022
' Copyright:    (c) Axle 2022
' Licence:      MIT No Attribution
'--------------------------------------------------------------------------------

' You can change the radius and pen size below.
Const RADIUS = 10  ' Do NOT exceed 20! 80 char MAX Console width.
Const PEN = 10  ' About 1:1 with the Radius to get single char.
Declare Function main_procedure() As Integer

main_procedure()

Function main_procedure() As Integer  ' Main procedure

    ' Defining a 2D array of array[Rows][Columns], initialized to empty.
    ' Radius *2 for circumference, *2 for double width character "O ", +1 for
safety = *5
    Dim As String canvas(RADIUS * 5, RADIUS * 5)
    ' Changing the following String characters will change the circle display.
    Dim As String Foreground = "O "  ' EX add space after chr to make circle (MAX
2 chrs)
    Dim As String Background = ". "  ' EX add space after chr to make circle (MAX
2 chrs)

    Const As Integer Radius = RADIUS
    Const As Integer Tolerance = PEN  ' EX Larger number will create a wider
drawing pen
```

```
    Dim As Integer Row_yy = 0  ' EX counters for array position
    Dim As Integer Col_xx = 0  ' EX counters for array position
    Dim As Integer Row_y  ' ST Counter range of circumference. -Radius to +radius
    Dim As Integer Col_x  ' ST Counter range of circumference. -Radius to +radius
    Dim As Integer equation
    For Row_y = -Radius To Radius Step 1  ' PR + EX
        Col_xx =0  ' EX reset the column counter for each row.
        For Col_x = -Radius To Radius Step 1  ' PR + EX
            ' Test if it is at the radius
            equation = Row_y*Row_y + Col_x*Col_x - Radius*Radius  ' EX
            If (Abs(equation) < Tolerance) Then 'ST
                'Print Foreground;
                canvas(Row_yy, Col_xx) = Left(Foreground, 2)  ' ST
            Else
                'Print Background;
                canvas(Row_yy, Col_xx) = Left(Background, 2)  ' ST
            End If

            'Print canvas(Row_yy, Col_xx);  ' ST
            Col_xx += 1  ' EX Increment Columns
        Next Col_x

        Row_yy += 1  ' EX Increment Rows
        'Print ""  ' ST
    Next Row_y

    ' Commenting out the loop below, and then enabling the
    ' printf statements above will print the circle directly
    ' rather than populating the array first and printing later :)
    ' Comment out (b) and enable (c) to print directly from the array/list.
    ' Or comment out (a)(c) and enable (b) to print directly without the
array/list.

    ' Print the rows and columns from our canvas array.
    ' The last increment of Row_yy and Col_xx from the loop above
    ' contains the correct lengths for the following loop.
    For Row_y = 0 To Row_yy Step 1  ' PR + EX
        For Col_x = 0 To Col_xx Step 1  ' PR + EX
            Print canvas(Row_y, Col_x);  ' ST
        Next Col_x
        Print ""  ' ST
    Next  Row_y

    Print "Press any key to continue..."
    Sleep  ' Sleep until a key is pressed
    Return 0  ' EX
End Function  ' END PR
```

Language: Python 3

Code: “Statements.py”

```
#-------------------------------------------------------------------------------
# Name:        Statements(ST), expressions(EX) and Procedures(PR)
# Purpose:     Example
#
# Platform:    REPL, Win64, Ubuntu64
#
```

```
# Author:      Axle
# Created:     31/01/2022
# Updated:     19/02/2022
# Copyright:   (c) Axle 2022
# Licence:     MIT No Attribution
#-------------------------------------------------------------------------------

# You can change the radius and pen size below.
RADIUS = 10  # Do NOT exceed 20! 80 char MAX Console width.
PEN = 10  # About 1:1 with the Radius to get single char.

def main():  # Main procedure

    # Defining a 2D array of array[Rows][Columns], initialized to empty.
    # Radius *2 for circumference, *2 for double width character "O ", +1 for
safety = *5
    canvas = [[None]* RADIUS * 5 for _ in range(RADIUS * 5)]
    # Changing the following String characters will change the circle display.
    Foreground = "O "  # EX add space after chr to make circle (MAX 2 chrs)
    Background = ". "  # EX add space after chr to make circle (MAX 2 chrs)

    Radius = RADIUS
    Tolerance = PEN  # EX Larger number will create a wider drawing pen

    Row_yy = 0  # EX counters for array position
    Col_xx = 0  # EX counters for array position
    Row_y = 0  # ST Counter range of circumference. -Radius to +radius
    Col_x = 0  # ST Counter range of circumference. -Radius to +radius
    equation = 0  # EX
    for Row_y in range(-Radius,  Radius +1):  # PR + EX
        Col_xx =0  # EX reset the column counter for each row.
        for Col_x in range(-Radius,  Radius +1):  # PR + EX
            # Test if it is at the radius
            equation = Row_y*Row_y + Col_x*Col_x - Radius*Radius  # EX
            if (abs(equation) < Tolerance):  # ST
                canvas[Row_yy][Col_xx] = Foreground[:2]  # ST
            else:
                canvas[Row_yy][Col_xx] = Background[:2]  # ST

            #print(canvas[Row_yy][Col_xx], end="")  # ST
            Col_xx += 1  # EX Increment Columns

        Row_yy += 1  # EX Increment Rows
        #print("")  # ST

    # Commenting out the loop below, and then enabling the
    # printf statements above will print the circle directly
    # rather than populating the array first and printing later :)
    # Comment out (b) and enable (c) to print directly from the array/list.
    # Or comment out (a)(c) and enable (b) to print directly without the
array/list.

    # Print the rows and columns from our canvas array.
    # The last increment of Row_yy and Col_xx from the loop above
    # contains the correct lengths for the following loop.
    for Row_y in range(0,  Row_yy):  # PR + EX
        for Col_x in range(0,  Col_xx):  # PR + EX
            print(canvas[Row_y][Col_x], end="")  # ST
```

```
        print("")  # ST

    input("Press [Enter] to exit.")
    return None  # EX
    # END PR

if __name__ == '__main__':
    main()
```

Decisions

If Then Else, Select Case, Switch Case, Ternary

Decisions control the flow of the application and redirect the program execution to different components within our main programs code structure and act like a switch in a similar way that a traffic light would direct the flow of traffic toward different streets.

Decisions are the brains and logic of an application directing the programming flow to the correct lines of code as required. Decisions and logic will make up a significant portion of our coding as this is the fundamental management component. Program logic can be simple or extremely complex if the solution requires it. Artificial Intelligence (AI) would be an example requiring very complic logic comparisons and switches.

Decisions will be made most commonly in the form of an *If true or false* comparison.

If (Comparison is True) **Then**
  Do something
Else If (Comparison is False) **Then**
  Do something else
Else Rem Comparisons were neither True nor False
  Do something if Comparisons were neither True nor False

Language: C

Code: “Decisions.c”

```
//------------------------------------------------------------------------------
// Name:        Decisions.c
// Purpose:     Example
// Title:       "A Snail's Life"
//
// Platform:    Win64, Ubuntu64
//
// Author:      Axle
// Created:     03/02/2022
// Updated:     19/02/2022
// Copyright:   (c) Axle 2022
// Licence:     MIT No Attribution
//------------------------------------------------------------------------------

// C std library headers.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <time.h>  // To seed Random.

// Platform specific headers.
// Test if Windows or Unix OS
#ifdef _WIN32
#define OS_Windows 1  // 1 = True (aka Bool)
#define OS_Unix 0
#endif

#ifdef __unix__  // __linux__
#define OS_Unix 1
#define OS_Windows 0  // 0 = False (aka Bool)
#endif

// Global Constants
#define ROWS 10
#define COLUMNS 40
#define STRMAX 4  // A buffer of 4 chars to hold our strings.

int UpdateScreen(char table[ROWS][COLUMNS][STRMAX]);
int Con_Clear(void);
void S_Pause(void);
int S_getchar(void);

int main()  // Main procedure
    {
    const int Row_start = ROWS - ROWS + 1;  // Set the boundaries inside of the
fence.
    const int Row_end = ROWS - 2;  // -2 To account for the fence at top and
bottom.
    const int Row_width = Row_end - Row_start;  // ROWS - 2 To account for the
fence at each side.
    const int Col_start = COLUMNS - COLUMNS + 1;  // Set the boundaries inside of
the fence.
    const int Col_end = COLUMNS - 2;  // -2 To account for the fence at each side.
    const int Col_width = Col_end - Col_start;  // COLUMNS - 2 To account for the
fence at each side.

    // Char 'r', 's' are not strings and must use the array[n][n][0] to be entered
    // as the first character of the string. The STRMAX allows us 4 char spaces
    // initialised to 0 to build a string in.
    // In C '' means a single character and "" represents a string.
    // array[n][n][STRMAX] = {'\0', '\0', '\0', '\0',}  // '\0' is null terminator
(end of string)
    // array[n][n][STRMAX] = {'v', '\0', '\0', '\0',}  // our string with 1
letter.
    // array[n][n] // to access it as a string "v".
    // Define the characters for the console game.
    const char chr_1 = ' ';
    const char chr_2 = '@';
    const char chr_3 = 'v';
    const char chr_4 = 'w';
    const char chr_5 = '+';

    // Values to random generate grass.
    // 0 - 10 (0=0% fill, 5= 50% fill, 10= 100% fill)
    const int GFill = 5;
    // 0 - 10 Split rand fill between % 0='w' and 10='W'.
    const int GSplit = 4;
```

```
    // defining a 2D array of array[Rows][Columns], initialised to empty.
    static char Table[ROWS][COLUMNS][STRMAX] = {'\0'};
    static int Sprite[2][2];  // Sprite[0][0|1] = current [Row|Col]
    int Row_y;  // Counter for rows.
    int Col_x;  // Counter for columns.

    // Use current time as seed for random generator
    srand(time(0));
    // Fill the array with some data.
    for (Row_y = 0; Row_y < ROWS; Row_y++)  // Count through each row.
        {
        for (Col_x = 0; Col_x < COLUMNS; Col_x++)  // Count through each column.
            {
            if ((Col_x >= Col_start) && (Col_x <= Col_end) && (Row_y >= Row_start)
&& (Row_y <= Row_end))
                {
                // Build a string with values for each cell.
                // rand()%10 returns 0 to 9
                if (rand() % 10 > GFill)  // Total grass GFill = 5(50%) fill.
                    {
                    if (rand() % 10 > GSplit)  //  Split between v and w.
                        {
                        Table[Row_y][Col_x][0] = chr_3;  // Insert a character for
grass.
                        }
                    else
                        {
                        Table[Row_y][Col_x][0] = chr_4;  // Insert a character for
grass.
                        }
                    Table[Row_y][Col_x][1] = '\0';
                    }
                else
                    {
                    Table[Row_y][Col_x][0] = chr_1;
                    Table[Row_y][Col_x][1] = '\0';
                    }
                }
            else
                {
                Table[Row_y][Col_x][0] = chr_5;  // Insert the fence character.
                Table[Row_y][Col_x][1] = '\0';
                }
            }
        }

    // Variables to track the Sprite (Snail) position.
    // Sprite[Sp_now][Sp_row] <- array format.
    const int Sp_now = 0;// Current sprite position.
    //const int Sp_last = 1;// previous position (Unused).
    const int Sp_row = 0;
    const int Sp_col = 1;

    // Set the snail at a random location and record it in an array.
    Sprite[Sp_now][Sp_row] = (rand() % Row_width) +1;  // +1 for left 0 boundary
alignment.
    Sprite[Sp_now][Sp_col] = (rand() % Col_width) +1;  // +1 for top 0 boundary
alignment..
```

```
    // Update the Table with the sprite.
    Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]][0] = chr_2;

    // Begin main movement and life counter loop
    // Note! that Control character 7 '\a' is system dependent and may not make a
sound.
    char Input;  // 'N''S''e''w' etc are single characters, not Strings here.
    int Lnrg = 10;  // Start at 10 life points. Reduces by 1 each move.
    while(1)
        {
        UpdateScreen(Table);  // Update the screen.
        printf("+ Life energy = %3d \n", Lnrg);  //  %3d allocates 3 positions.
        if (Lnrg <= 0)
            {
            printf("\a");  // System bell (ASCII Control Chr 7)
            printf("Oh No! You are out of life energy...\n");
            break;
            }

        printf("N,S,E,W to move : Q to Quit\n");
        printf("Type your selection followed by [Enter] >>");
        Input = S_getchar();
        printf("\n");
        Lnrg -= 1;  // Reduce 1 nrg point for each move including hitting the
boundary.

        if ((Input == 'N') || (Input == 'n'))
            {
            if (Sprite[Sp_now][Sp_row] - 1 != Row_start -1)
                {
                // The following if else block would be best subed to a function
                // to limit duplication.
                if (Table[Sprite[Sp_now][Sp_row] -1][Sprite[Sp_now][Sp_col]][0] ==
chr_3)
                    {
                    Lnrg +=1;  // Increase 1 nrg point for small grass.
                    }
                else if (Table[Sprite[Sp_now][Sp_row] -
1][Sprite[Sp_now][Sp_col]][0] == chr_4)
                    {
                    Lnrg +=2;  // Increase 2 nrg point for large grass.
                    }
                else
                    {
                    // pass
                    }
                Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]][0] = chr_1;
                Sprite[Sp_now][Sp_row] -=1;
                Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]][0] = chr_2;
                // Move Lnrg -= 1; here to not reduce Life nrg when hitting the
boundary.
                }
            else
                {
                printf("\a");  // System bell (ASCII Control Chr 7)
                }

            }
```

```
        else if ((Input == 'S') || (Input == 's'))
            {
            if (Sprite[Sp_now][Sp_row] + 1 != Row_end +1)
                {
                if (Table[Sprite[Sp_now][Sp_row] +1][Sprite[Sp_now][Sp_col]][0] ==
chr_3)
                    {
                    Lnrg +=1;  // Increase 1 nrg point for small grass.
                    }
                else if (Table[Sprite[Sp_now][Sp_row]
+1][Sprite[Sp_now][Sp_col]][0] == chr_4)
                    {
                    Lnrg +=2;  // Increase 2 nrg points for large grass.
                    }
                else
                    {
                    // pass
                    }
                Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]][0] = chr_1;
                Sprite[Sp_now][Sp_row] +=1;
                Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]][0] = chr_2;
                // Move Lnrg -= 1; here to not reduce Life nrg when hitting the
boundary.
                }
            else
                {
                printf("\a");  // System bell (ASCII Control Chr 7)
                }
            }
        else if ((Input == 'E') || (Input == 'e'))
            {
            if (Sprite[Sp_now][Sp_col] + 1 != Col_end +1)
                {
                if (Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col] +1][0] ==
chr_3)
                    {
                    Lnrg +=1;  // Increase 1 nrg point for small grass.
                    }
                else if (Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]
+1][0] == chr_4)
                    {
                    Lnrg +=2;  // Increase 2 nrg points for large grass.
                    }
                else
                    {
                    // pass
                    }
                Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]][0] = chr_1;
                Sprite[Sp_now][Sp_col] +=1;
                Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]][0] = chr_2;
                // Move Lnrg -= 1; here to not reduce Life nrg when hitting the
boundary.
                }
            else
                {
                printf("\a");  // System bell (ASCII Control Chr 7)
                }
            }
        else if ((Input == 'W') || (Input == 'w'))
```

```
            {
            if (Sprite[Sp_now][Sp_col] - 1 != Col_start -1)
                {
                if (Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col] -1][0] ==
chr_3)
                    {
                    Lnrg +=1;  // Increase 1 nrg point for small grass.
                    }
                else if (Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col] -
1][0] == chr_4)
                    {
                    Lnrg +=2;  // Increase 2 nrg points for large grass.
                    }
                else
                    {
                    // pass
                    }
                Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]][0] = chr_1;
                Sprite[Sp_now][Sp_col] -=1;
                Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]][0] = chr_2;
                // Move Lnrg -= 1; here to not reduce Life nrg when hitting the
boundary.
                }
            else
                {
                printf("\a");  // System bell (ASCII Control Chr 7)
                }
            }
        else if ((Input == 'Q') || (Input == 'q'))
            {
            break;
            }
        else
            {
            // Pass
            }
        }

    S_Pause();
    return 0;
    }

int UpdateScreen(char table[ROWS][COLUMNS][STRMAX])
    {
    Con_Clear();
    int Row_y;  // Counter for rows.
    int Col_x;  // Counter for columns.

    // Print the array to the console.
    for (Row_y = 0; Row_y < ROWS; Row_y++)// Count through each row.
        {
        for (Col_x = 0; Col_x < COLUMNS; Col_x++)  // Count through each column.
            {
            printf("%s", table[Row_y][Col_x]);  // No line breaks.
            }
        printf("\n");  // Print a line break to start next row.
        }
    return 0;
    }
```

```
int Con_Clear(void)
    {
    // The system() call allows the programmer to run OS command line batch
commands.
    // It is discouraged as there are more appropriate C functions for most tasks.
    // I am only using it in this instance to avoid invoking additional OS API
headers and code.
    if (OS_Windows)
        {
        system("cls");
        }
    else if (OS_Unix)
        {
        system("clear");
        }
    return 0;
    }

// Safe Pause
void S_Pause(void)
    {
    // This function is referred to as a wrapper for S_getchar()
    printf("Press any key to continue...");
    S_getchar();  // Uses S_getchar() for safety.
    }

// Safe getcar() removes all artefacts from the stdin buffer.
int S_getchar(void)
    {
    // This function is referred to as a wrapper for getchar()
    int i = 0;
    int ret;
    int ch;
    // The following enumerates all characters in the buffer.
    while((ch = getchar()) != '\n' && ch != EOF )
        {
        // But only keeps and returns the first char.
        if (i < 1)
            {
            ret = ch;
            }
        i++;
        }
    return ret;
    }
```

Language: FreeBASIC

Code: “Decisions.bas”

```
'--------------------------------------------------------------------------------
' Name:         Decisions.bas
' Purpose:      Example
' Title:        "A Snail’s Life"
'
' Platform:     Win64, Ubuntu64
'
```

```
' Author:      Axle
' Created:     03/02/2022
' Updated:     19/02/2022
' Copyright:   (c) Axle 2022
' Licence:     MIT No Attribution
'-------------------------------------------------------------------------------


#ifdef __FB_WIN32__
#define OS_Windows 1
#define OS_Unix 0
#endif

#ifdef __FB_UNIX__'__FB_LINUX__
' TODO
#define OS_Unix 1
#define OS_Windows 0
#endif

' Global Constants
#define ROWS 10
#define COLUMNS 40

Declare Function UpdateScreen(table() As String) As Integer
Declare Function Con_Clear() As Integer
Declare Function Con_Pause() As Integer

Declare Function main_procedure() As Integer
main_procedure()

Function main_procedure() As Integer  ' Main procedure

    '#ifdef OS_Windows
    ' Windows code
    '#else
    ' GNU/Linux code
    '#endif
    Const As Integer Row_start = ROWS - ROWS + 1  ' Set the boundaries inside of
the fence.
    Const As Integer Row_end = ROWS - 2  ' -2 To account for the fence at top and
bottom.
    Const As Integer Row_width = Row_end - Row_start  ' ROWS - 2 To account for
the fence at each side.
    Const As Integer Col_start = COLUMNS - COLUMNS + 1  ' Set the boundaries
inside of the fence.
    Const As Integer Col_end = COLUMNS - 2  ' -2 To account for the fence at each
side.
    Const As Integer Col_width = Col_end - Col_start  ' COLUMNS - 2 To account for
the fence at each side.

    ' Similar to the C example I am using characters as their integer value (See
ASCII Char Chart).
    ' I am converting a single string character to its Asc-ii value. aka integer.
    ' Asc("a"), Chr(97)
    ' Define the characters for the console game.
    Const As Integer chr_1 = Asc(" ")  ' Dec 32
    Const As Integer chr_2 = Asc("@")  ' Dec 64
    Const As Integer chr_3 = Asc("v")  ' Dec 118
    Const As Integer chr_4 = Asc("w")  ' Dec 119
```

```
    Const As Integer chr_5 = Asc("+")  ' Dec 43

    ' Values to randomly generate grass.
    ' 0 - 10 (0=0% fill, 5= 50% fill, 10= 100% fill)
    Const As Integer GFill = 5
    ' 0 - 10 Split rand fill between % 0='w' and 10='W'.
    Const As Integer GSplit = 4

    ' Defining a 2D array of array[Rows][Columns], initialized to empty.
    Static As String Table(ROWS, COLUMNS)
    Static As Integer Sprite(2, 2)  ' Sprite[0][0|1] = current [Row|Col]
    Dim As Integer Row_y  ' Counter for rows.
    Dim As Integer Col_x  ' Counter for columns.

    ' Use current time as seed for the random generator.
    Randomize  ' without arguments defaults to system time.
    ' Fill the array with some data.
    For Row_y = 0 To ROWS -1 Step 1  ' Count through each row.
        For Col_x = 0 To COLUMNS -1 Step 1  ' Count through each column.
            If ((Col_x >= Col_start) And (Col_x <= Col_end) And (Row_y >=
Row_start) And (Row_y <= Row_end)) Then
                ' Build a string with values for each cell.
                ' Int(Rnd * 10) returns 0 to 9
                If (Int(Rnd * 10) > GFill) Then  ' Total grass GFill = 5(50%)
fill.
                    If (Int(Rnd * 10) > GSplit) Then  '  Split between v and w.
                        Table(Row_y, Col_x) = Chr(chr_3)  ' Returns as a formatted
string.
                    Else
                        Table(Row_y, Col_x) = Chr(chr_4)  ' Insert a character for
grass.
                    End If
                Else
                    Table(Row_y, Col_x) = Chr(chr_1)  ' Insert a space character.
                End If
            Else
                Table(Row_y, Col_x) = Chr(chr_5)  ' Insert the fence character.
            End If
        Next Col_x
    Next Row_y

    ' Variables to track the Sprite (Snail) position.
    ' Sprite[Sp_now][Sp_row] <- array format.
    Const As Integer Sp_now = 0  ' Current sprite position.
    'Const As Integer Sp_last = 1  ' previous position (Unused).
    Const As Integer Sp_row = 0
    Const As Integer Sp_col = 1

    ' Set the snail at a random location and record it in an array.
    Sprite(Sp_now, Sp_row) = Int(Rnd * Row_width) +1  ' +1 for left 0 boundary.
    Sprite(Sp_now, Sp_col) = Int(Rnd * Col_width) +1  ' +1 for top 0 boundary.

    ' Update the Table with the sprite.
    Table(Sprite(Sp_now, Sp_row), Sprite(Sp_now, Sp_col)) = Chr(chr_2)

    ' Begin main movement and life counter loop.
    ' Note! that Control character 7 '\a' is system dependent and may not make a
sound.
    Dim As String Inputs  ' 'N''S''e''w'
```

```
    Dim As Integer Lnrg = 10  ' Start at 10 life points. Reduces by 1 each move.
    While(1)
        UpdateScreen(Table())  ' Update the screen.
        Print "+ Life energy= " & Left(Str(Lnrg), 3)  ' Left( , 3) allocates 3
positions.
        If (Lnrg <= 0) Then
            Print !"\a"  ' System bell (ASCII Control Chr 7)
            Print "Oh No! You are out of life energy..."
            Exit While
        End If
        Print "N,S,E,W to move : Q to Quit"
        Input "Type your selection followed by [Enter] >>", Inputs
        Print ""
        Lnrg -= 1  ' Reduce 1 nrg point for each move including hitting the
boundary.

        If (Left(Inputs, 1) = "N") Or (Left(Inputs, 1) = "n") Then
            If (Sprite(Sp_now, Sp_row) - 1 <> Row_start -1) Then
                ' The following if else block would be best subed to a function
                ' to limit duplication.
                If (Table(Sprite(Sp_now, Sp_row) -1, Sprite(Sp_now, Sp_col)) =
Chr(chr_3)) Then
                    Lnrg +=1  ' Increase 1 nrg point for small grass.
                Elseif (Table(Sprite(Sp_now, Sp_row) -1, Sprite(Sp_now, Sp_col)) =
Chr(chr_4)) Then
                    Lnrg +=2  ' Increase 2 nrg points for large grass.
                Else
                    ' pass
                End If
                Table(Sprite(Sp_now, Sp_row), Sprite(Sp_now, Sp_col)) = Chr(chr_1)
                Sprite(Sp_now, Sp_row) -=1
                Table(Sprite(Sp_now, Sp_row), Sprite(Sp_now, Sp_col)) = Chr(chr_2)
                ' Move Lnrg -= 1; here to not reduce Life nrg when hitting the
boundary.
            Else
                Print !"\a"  ' System bell (ASCII Control Chr 7)
            End If
        Elseif (Left(Inputs, 1) = "S") Or (Left(Inputs, 1) = "s") Then
            If (Sprite(Sp_now, Sp_row) + 1 <> Row_end +1) Then
                If (Table(Sprite(Sp_now, Sp_row) +1, Sprite(Sp_now, Sp_col)) =
Chr(chr_3)) Then
                    Lnrg +=1  ' Increase 1 nrg point for small grass.
                Elseif (Table(Sprite(Sp_now, Sp_row) +1, Sprite(Sp_now, Sp_col)) =
Chr(chr_4)) Then
                    Lnrg +=2  ' Increase 2 nrg points for large grass.
                Else
                    ' pass
                End If
                Table(Sprite(Sp_now, Sp_row), Sprite(Sp_now, Sp_col)) = Chr(chr_1)
                Sprite(Sp_now, Sp_row) +=1
                Table(Sprite(Sp_now, Sp_row), Sprite(Sp_now, Sp_col)) = Chr(chr_2)
                ' Move Lnrg -= 1; here to not reduce Life nrg when hitting the
boundary.
            Else
                Print !"\a"  ' System bell (ASCII Control Chr 7)
            End If
        Elseif (Left(Inputs, 1) = "E") Or (Left(Inputs, 1) = "e") Then
            If (Sprite(Sp_now, Sp_col) + 1 <> Col_end +1) Then
                If (Table(Sprite(Sp_now, Sp_row), Sprite(Sp_now, Sp_col) +1) =
```

```
Chr(chr_3)) Then
                    Lnrg +=1  ' Increase 1 nrg point for small grass.
                Elseif (Table(Sprite(Sp_now, Sp_row), Sprite(Sp_now, Sp_col) +1) =
Chr(chr_4)) Then
                    Lnrg +=2  ' Increase 2 nrg points for large grass.
                Else
                    ' pass
                End If
                Table(Sprite(Sp_now, Sp_row), Sprite(Sp_now, Sp_col)) = Chr(chr_1)
                Sprite(Sp_now, Sp_col) +=1
                Table(Sprite(Sp_now, Sp_row), Sprite(Sp_now, Sp_col)) = Chr(chr_2)
                ' Move Lnrg -= 1; here to not reduce Life nrg when hitting the
boundary.
            Else
                Print !"\a"  ' System bell (ASCII Control Chr 7)
            End If
        Elseif (Left(Inputs, 1) = "W") Or (Left(Inputs, 1) = "w") Then
            If (Sprite(Sp_now, Sp_col) - 1 <> Col_start -1) Then
                If (Table(Sprite(Sp_now, Sp_row), Sprite(Sp_now, Sp_col) -1) =
Chr(chr_3)) Then
                    Lnrg +=1  ' Increase 1 nrg point for small grass.
                Elseif (Table(Sprite(Sp_now, Sp_row), Sprite(Sp_now, Sp_col) -1) =
Chr(chr_4)) Then
                    Lnrg +=2  ' Increase 2 nrg points for large grass.
                Else
                    ' pass
                End If
                Table(Sprite(Sp_now, Sp_row), Sprite(Sp_now, Sp_col)) = Chr(chr_1)
                Sprite(Sp_now, Sp_col) -=1
                Table(Sprite(Sp_now, Sp_row), Sprite(Sp_now, Sp_col)) = Chr(chr_2)
                ' Move Lnrg -= 1; here to not reduce Life nrg when hitting the
boundary.
            Else
                Print !"\a"  ' System bell (ASCII Control Chr 7)
            End If
        Elseif (Left(Inputs, 1) = "Q") Or (Left(Inputs, 1) = "q") Then
            Exit While
        Else
            ' Pass
        End If

    Wend

    Con_Pause()
    Return 0
End Function  ' END main_procedure <---

Function UpdateScreen(table() As String) As Integer
    Con_Clear()
    Dim As Integer Row_y  ' Counter for rows.
    Dim As Integer Col_x  ' Counter for columns.

    ' Print the array to the console.
    For Row_y = 0 To ROWS -1 Step 1  ' Count through each row.
        For Col_x = 0 To COLUMNS -1 Step 1  ' Count through each column.
            Print table(Row_y, Col_x);  ' No line breaks.
        Next Col_x
        Print ""  ' Print a line break to start next row.
    Next Row_y
```

```
    Return 0
End Function

Function Con_Clear() As Integer
    ' The system() call allows the programmer to run OS command line batch
commands.
    ' It is discouraged as there are more appropriate C functions for most tasks.
    ' I am only using it in this instance to avoid invoking additional OS API
headers and code.
    If (OS_Windows) Then
        Shell "cls"
    Elseif (OS_Unix) Then
        Shell "clear"
    End If
    Return 0
End Function

Function Con_Pause() As Integer
    Dim As Integer dummy
    Input "Press [Enter] key to continue...", dummy

    Return 0
End Function
```

Language: Python 3

Code: “Decisions.py”

```
#--------------------------------------------------------------------------------
# Name:         Decisions.py
# Purpose:      Example
# Title:        "A Snail’s Life"
#
# Platform:     REPL*, Win64, Ubuntu64
#
# Author:       Axle
# Created:      03/02/2022
# Updated:      19/02/2022
# Copyright:    (c) Axle 2022
# Licence:      MIT No Attribution
#--------------------------------------------------------------------------------
#
# NOTE! Lift the divider on the Python Interpreter output console to see the
# full game in REPL.
#
# * This example is best run in a native OS console window.
#
# ' \' after a line of code is a line continuation character and allows you to
# split long lines of code over several lines. You can use a comment after a
# continuation character as long as there is no space /#comment.
#--------------------------------------------------------------------------------

import random
```

```
# Global Constants
ROWS  = 10
COLUMNS = 40

def main():

    if 1 == Con_IsREPL():  # test if we are in Python 3 REPL.
        print("This application is best viewed from the OS Command interpreter")
        Con_Pause()

    global ROWS, COLUMNS
    Row_start = ROWS - ROWS + 1  # Set the boundaries inside of the fence.
    Row_end = ROWS - 2  # -2 To account for the fence at top and bottom.
    Row_width = Row_end - Row_start  # ROWS - 2 for the fence.
    Col_start = COLUMNS - COLUMNS + 1  # Set the boundaries inside of the fence.
    Col_end = COLUMNS - 2  # -2 To account for the fence at each side.
    Col_width = Col_end - Col_start  #   COLUMNS - 2 for the fence.

    #  I am using strings instead of characters.
    #  Define the characters for the console game.
    chr_1 = " "
    chr_2 = "@"
    chr_3 = "v"
    chr_4 = "w"
    chr_5 = "+"

    # Values to random generate grass.
    # 0 - 10 (0=0% fill, 5= 50% fill, 10= 100% fill)
    GFill = 5
    # 0 - 10 Split rand fill between % 0='w' and 10='W'.
    GSplit = 4

    # defining a 2D array of array[Rows][Columns], initialized to empty.
    # Note! Columns is inside Rows.
    Table = [[None]* COLUMNS for _ in range(0, ROWS)]
    Sprite = [[None]* 2 for _ in range(0, 2)]  # Sprite[0][0|1] = current
[Row|Col]
    Row_y = 0  # Counter for rows.
    Col_x = 0  # Counter for columns.

    # Use current time as seed for random generator.
    # random.randint() uses OS_Rand or time as a seed. We don't need to create a
    # seed as we do in C or FreeBASIC.
    # Fill the array with some data.
    for Row_y in range(0, ROWS ):  # Count through each row.
        for Col_x in range(0, COLUMNS ):  # Count through each column.
            if ((Col_x >= Col_start) and (Col_x <= Col_end) \#LineContinue
                and (Row_y >= Row_start) and (Row_y <= Row_end)):
                # Build a string with values for each cell.
                # random.randint(0, 9) returns 0 to 9
                if (random.randint(1, 9) > GFill):  # Total grass GFill = 5(50%)
fill.
                    if (random.randint(1, 9) > GSplit):  #  Split between v and w.
                        Table[Row_y][Col_x] = chr_3  # Insert a character for
grass.
                    else:
                        Table[Row_y][Col_x] = chr_4  # Insert a character for
grass.
                else:
```

```
                    Table[Row_y][Col_x] = chr_1  # Insert a space character.
            else:

                Table[Row_y][Col_x] = chr_5  # Insert the fence character.

    # Variables to track the Sprite (Snail) position.
    # Sprite[Sp_now][Sp_row] <- array format.
    Sp_now = 0  # Current sprite position.
    #Const As Integer Sp_last = 1  # previous position (Unused).
    Sp_row = 0
    Sp_col = 1

    # Set the snail at a random location and record it in an array.
    # "Row_width -1" is to keep it in line with C and FB Rand generated range.
    Sprite[Sp_now][Sp_row] = random.randint(0, Row_width -1) +1  # +1 for left 0
boundary alignment..
    Sprite[Sp_now][Sp_col] = random.randint(0, Col_width -1) +1  # +1 for top 0
boundary alignment..

    # Update the Table with the sprite.
    Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]] = chr_2

    # Begin main movement and life counter loop
    # Note! that Control character 7 '\a' is system dependent and may not make a
sound.
    Inputs = ""  # 'N''S''e''w'
    Lnrg = 10  # Start at 10 life points. Reduces by 1 each move.
    while(1):
        UpdateScreen(Table)  # Update the screen.
        print("+ Life energy= " + str(Lnrg)[0:3])  # String[0:3] allocates 3
positions.
        if (Lnrg <= 0):
            #print("\a", end="")  # System bell (ASCII Control Chr 7) Con_bell()
            Con_bell()
            print("Oh No! You are out of life energy...")
            break
        print("N,S,E,W to move : Q to Quit")
        Inputs = input("Type your selection followed by [Enter] >>")
        print("")
        Lnrg -= 1  # Reduce 1 nrg point for each move including hitting the
boundary.

        if (Inputs[0:1] == "N") or (Inputs[0:1] == "n"):
            if (Sprite[Sp_now][Sp_row] - 1 != Row_start -1):
                # The following if else block would be best subed to a function
                # to limit duplication.
                if (Table[Sprite[Sp_now][Sp_row] -1][Sprite[Sp_now][Sp_col]] ==
chr_3):
                    Lnrg +=1  # Increase 1 nrg point for small grass.
                elif (Table[Sprite[Sp_now][Sp_row] -1][Sprite[Sp_now][Sp_col]] ==
chr_4):
                    Lnrg +=2  # Increase 2 nrg points for large grass.
                else:
                    pass
                Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]] = chr_1
                Sprite[Sp_now][Sp_row] -=1
                Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]] = chr_2
                # Move Lnrg -= 1; here to not reduce Life nrg when hitting the
boundary.
```

```
            else:
                #print("\a", end="") # System bell (ASCII Control Chr 7)
                Con_bell()
        elif (Inputs[0:1] == "S") or (Inputs[0:1] == "s"):
            if (Sprite[Sp_now][Sp_row] + 1 != Row_end +1):
                if (Table[Sprite[Sp_now][Sp_row] +1][Sprite[Sp_now][Sp_col]] ==
chr_3):
                    Lnrg +=1  # Increase 1 nrg point for small grass.
                elif (Table[Sprite[Sp_now][Sp_row] +1][Sprite[Sp_now][Sp_col]] ==
chr_4):
                    Lnrg +=2  # Increase 2 nrg points for large grass.
                else:
                    pass
                Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]] = chr_1
                Sprite[Sp_now][Sp_row] +=1
                Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]] = chr_2
                # Move Lnrg -= 1; here to not reduce Life nrg when hitting the
boundary.
            else:
                #print("\a", end="") # System bell (ASCII Control Chr 7)
                Con_bell()
        elif (Inputs[0:1] == "E") or (Inputs[0:1] == "e"):
            if (Sprite[Sp_now][Sp_col] + 1 != Col_end +1):
                if (Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col] +1] ==
chr_3):
                    Lnrg +=1  # Increase 1 nrg point for small grass.
                elif (Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col] +1] ==
chr_4):
                    Lnrg +=2  # Increase 2 nrg points for large grass.
                else:
                    pass
                Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]] = chr_1
                Sprite[Sp_now][Sp_col] +=1
                Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]] = chr_2
                # Move Lnrg -= 1; here to not reduce Life nrg when hitting the
boundary.
            else:
                #print("\a", end="")  # System bell (ASCII Control Chr 7)
                Con_bell()
        elif (Inputs[0:1] == "W") or (Inputs[0:1] == "w"):
            if (Sprite[Sp_now][Sp_col] - 1 != Col_start -1):
                if (Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col] -1] ==
chr_3):
                    Lnrg +=1  # Increase 1 nrg point for small grass.
                elif (Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col] -1] ==
chr_4):
                    Lnrg +=2  # Increase 2 nrg points for large grass.
                else:
                    pass
                Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]] = chr_1
                Sprite[Sp_now][Sp_col] -=1
                Table[Sprite[Sp_now][Sp_row]][Sprite[Sp_now][Sp_col]] = chr_2
                # Move Lnrg -= 1; here to not reduce Life nrg when hitting the
boundary.
            else:
                #print("\a", end="")  # System bell (ASCII Control Chr 7)
                Con_bell()
        elif (Inputs[0:1] == "Q") or (Inputs[0:1] == "q"):
            break
```

```
        else:
            pass

    Con_Pause()
    return None
# END main <---

# --> START helper functions

# Prints updated array to screen.
def UpdateScreen(table):
    Con_Clear()
    Row_y = 0  # Counter for rows.
    Col_x = 0  # Counter for columns.

    # Print the array to the console.
    for Row_y in range(0, ROWS):  # Count through each row.
        for Col_x in range(0, COLUMNS):  # Count through each column.
            print(table[Row_y][Col_x], end="") #  No line breaks.
        print("")  # Print a line break to start next row.
    return None

# Test if we are inside of the REPL interactive interpreter.
# This function is in alpha and may not work as expected.
def Con_IsREPL():
    import os
    if os.sys.stdin and os.sys.stdin.isatty():
        if os.isatty(os.sys.stdout.fileno()):
            return 0  # OS Command Line
        else:
            return 1  # REPL - Interactive Linux?
    else:
        return 1  # REPL - Interactive Windows?
    return None

# Console bell. May not work on all systems.
def Con_bell():
    import os
    if os.sys.stdin and os.sys.stdin.isatty():
        if os.isatty(os.sys.stdout.fileno()):
            # OS Command Line
            # Note! System bell may not work on all OSs.
            print("\a", end='')  # System ding (If enabled)
            return None
        else:
            pass  # REPL - Interactive Linux?
    else:
        pass  # REPL - Interactive Windows?
    return None

# Cross platform console clear.
# This function is in alpha and may not work as expected.
def Con_Clear():
    # The system() call allows the programmer to run OS command line batch
commands.
    # It is discouraged as there are more appropriate C functions for most tasks.
    # I am only using it in this instance to avoid invoking additional OS API
headers and code.
    import os
```

```
    if os.isatty(os.sys.stdout.fileno()):  # Clear function doesn't work in Py
REPL
        # for windows
        if os.name == 'nt':
            os.system('cls')
        # for mac and linux
        elif os.name == 'posix':
            os.system('clear')
        else:
            return -1  # Unknown OS
    else:
        return None

# Console Pause wrapper.
def Con_Pause():
    dummy = ""
    dummy = input("Press [Enter] key to continue...")

    return None

if __name__ == '__main__':
    main()
```

Loops

*Loop Until, For Next, While WEnd, Do Until, Do While, For In Next, Goto**

Loops are the powerhouse of programming and allow us to perform repetitive tasks.

I have already made use of loops in the previous examples as it is difficult to perform any coding without using the 8 principle components.

The evil GOTO statement. GOTO loops are often referred to as evil, never to be spoken of and if you should ever speak of it fire and lightning will rain down upon you… But what is the GOTO LABEL: statement anyway?

A GOTO LABEL: is a fundamental principle that exists in all computer code at the CPU level. It is unavoidable and part of the very structure that allows a computer to carry out the billions of tasks required of it every second. In Assembly language this is known as a Jump statement and is written in a similar form to the way it is found in low level languages such as C. Assembly uses several implementations in the form of JMP LABEL: (In C this would be GOTO LABEL:). It is the fundamental statement required to build loops in any programming language. In higher level languages GOTO Jumps are created in a more refined set of functions such as the For() loop and While() loops. GOTO is strongly discouraged in high level languages, and even removed from many, as the for and while loops are far safer and more appropriate in the context of those languages, but there is nothing inherently evil about GOTO other than its misuse. There are, although limited, occasions where the use of a GOTO LABEL: is appropriate, for example when we want to override the default behaviour of a programming language at a lower level.

GOTO is a little bit like underwear, we all make use of it, but it is generally hidden under a set of more appropriate clothing. We would be unlikely to walk down the street in our underwear as we

have more refined ways of implementing its use. But there is nothing inherently evil about underwear.

I have offered some code in the C example below to highlight how the GOTO function works.

Loops examples part a.

Language: C

Code: "Loopsa.c"

```c
//--------------------------------------------------------------------------------
// Name:        Loops.c
// Purpose:     Example
//
// Platform:    Win64, Ubuntu64
//
// Author:      Axle
// Created:     15/02/2022
// Updated:     18/02/2022
// Copyright:   (c) Axle 2022
// Licence:     MIT No Attribution
//--------------------------------------------------------------------------------

#include <stdio.h>
//#include <stdlib.h>
//#include <string.h>

// Define extra functions so we can place them at the bottom of the page.
void S_Pause(void);
int S_getchar(void);

int main()  // Main procedure
    {
    // A simple example of nested loops using for and while.
    const int num = 5;  // length of loops.

    // Set variables for enumerating (Counting through) the loops.
    int a = 0;
    int b = 0;
    int c = 0;
    // Loop level 1.
    for(a = 0; a <= num; a++)
        {
        printf("a%d:\n", a);
        // Loop level 2 (nested).
        for(b = 0; b <= a; b++)
            {
            printf("b%d:", b);
            // Loop level 3 (nested).
            for(c = 0; c <= b; c++)
                {
                printf("%d,", c);
                }
            printf("\n");
            }
        printf("\n");
```

```
        }

    S_Pause();

    // Reset our counters to zero.
    a = 0;
    b = 0;
    c = 0;
    // Loop level 1.
    while(a <= num)
        {
        printf("a%d:\n", a);
        // Loop level 2.
        while(b <= a)
            {
            printf("b%d:", b);
            // Loop level 3.
            while(c <= b)
                {
                printf("%d,", c);
                c++;
                }
            printf("\n");
            c = 0;
            b++;
            }
        printf("\n");
        b = 0;
        a++;
        }

    S_Pause();

    /* How loops are created in assembly.
    // Intel asm syntax
    // jmp label: is the equivalent of GOTO LABEL:

            mov eax, 0      ; set a counter to enumerate the loop
            mov ebx, num    ; store var num in ecx register
        top:                ; Label
            cmp eax, ebx    ; Test if eax == num
            je bottom       ; loop exit when while condition True
            BODY            ; ... Code inside the loop
            inc eax         ; Increment the counter +1
            jmp top         ; Jump to label top:
        bottom:             ; Label
    */

    a = 0;  // reset counter a to 0.
    // The while loop will print to 4 as the truth test is before the printf().
    // This is the correct way to implement a loop by using while or for.
    while(1)
        {
        if(a == num)
            {
            break;
            }
        printf("%d\n", a);
        a++;
```

```
        }

    /*
    // Assembly output for the above while loop.
    // Intel asm syntax
    .LC0:
            .string "%d\n"
    main:
            push    rbp
            mov     rbp, rsp
            sub     rsp, 16
            mov     DWORD PTR [rbp-8], 5
            mov     DWORD PTR [rbp-4], 0
    .L4:
            mov     eax, DWORD PTR [rbp-4]
            cmp     eax, DWORD PTR [rbp-8]
            je      .L7
            mov     eax, DWORD PTR [rbp-4]
            mov     esi, eax
            mov     edi, OFFSET FLAT:.LC0
            mov     eax, 0
            call    printf
            add     DWORD PTR [rbp-4], 1
            jmp     .L4
    .L7:
            nop
            mov     eax, 0
            leave
            ret
    */

    S_Pause();

    // Recreate a while loop (correctly) using goto (jmp)
    // This is effectively the same code as the while loop above.
    // The use of goto label is strongly discouraged unless there is
    // a specific requirement where a while or for loop is not available.
    // Start while loop
    int d = 0;
    goto L2;  // enter while loop.
L2:
    if(d == num)  // Truth test
        {
        goto L7;  // break out of loop
        }
    printf("%d\n", d);
    d++;  // Increment d.
    goto L2;
L7:  // exit while loop

    /*
    // Exact assembly output for the  goto (while) loop.
    // Intel asm syntax
    // eax, esi, etc are CPU registers.
    // Essentially the internal commands (keywords) or instruction set of the CPU.
    .LC0:
        .string "%d,\n"
    main:                                   ; main()
        push    rbp                         ; main()set integers
```

```
        mov     rbp, rsp                ; main() sets integers
        sub     rsp, 16                 ; main() sets integers to 16bit
        mov     DWORD PTR [rbp-8], 5    ; Declare variable num = 5
        mov     DWORD PTR [rbp-4], 0    ; Declare variable d = 0
        nop
    .L2:                                ; Start while loop
        mov     eax, DWORD PTR [rbp-4]
        cmp     eax, DWORD PTR [rbp-8]  ; Truth test
        je      .L7                     ; if True exit loop
        mov     eax, DWORD PTR [rbp-4]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf                  ; print d
        add     DWORD PTR [rbp-4], 1    ; Increment d (+1)
        jmp     .L2                     ; Loop
    .L7:                                ; Exit while loop
        nop
        mov     eax, 0
        leave                           ; Exit main()
        ret                             ; main() return statement
    */

    S_Pause();
    return 0;
    }

// --> START helper functions
// Safe Pause
void S_Pause(void)
    {
    // This function is referred to as a wrapper for S_getchar()
    printf("Press any key to continue...");
    S_getchar();// Uses S_getchar() for safety.
    }

// Safe getcar() removes all artefacts from the stdin buffer.
int S_getchar(void)
    {
    // This function is referred to as a wrapper for getchar()
    int i = 0;
    int ret;
    int ch;
    // The following enumerates all characters in the buffer.
    while((ch = getchar()) != '\n' && ch != EOF )
        {
        // But only keeps and returns the first char.
        if (i < 1)
            {
            ret = ch;
            }
        i++;
        }
    return ret;
    }
```

Language: FreeBASIC

Code: “Loopsa.bas”

```
'--------------------------------------------------------------------------------
' Name:         Loops.bas
' Purpose:      Example
'
' Platform:     Win64, Ubuntu64
'
' Author:       Axle
' Created:      15/02/2022
' Updated:      19/02/2022
' Copyright:    (c) Axle 2022
' Licence:      MIT No Attribution
'--------------------------------------------------------------------------------

' Define extra functions so we can place them at the bottom of the page.
Declare Function main_procedure() As Integer
Declare Function Con_Pause() As Integer

main_procedure()

Function main_procedure() As Integer ' Main procedure

    ' A simple example of nested loops using for and while.
    Const As Integer num = 5 '  length of loops.

    ' Set variables for enumerating (Counting through) the loops.
    Dim As Integer a = 0
    Dim As Integer b = 0
    Dim As Integer c = 0
    ' Loop level 1.
    For a = 0 To num Step 1
        Print "a" & a & ":"
        ' Loop level 2 (nested).
        For b = 0 To a Step 1
            Print "b" & b & ":";
            ' Loop level 3 (nested).
            For c = 0 To b Step 1
                Print  c & ",";
            Next c
            Print ""
        Next b
        Print ""
    Next a

    'S_Pause()
    Con_Pause()

    ' Reset our counters to zero.
    a = 0
    b = 0
    c = 0
    ' Loop level 1.
    While(a <= num)
        Print "a" & a & ":"
        ' Loop level 2 (nested).
        While(b <= a)
            Print "b" & b & ":";
            ' Loop level 3 (nested).
            While(c <= b)
```

```
                Print c & ",";
                c+=1
            Wend
            Print ""
            c = 0
            b+=1
        Wend
        Print ""
        b = 0
        a+=1
    Wend

    Con_Pause()

    /' How loops are created in assembly.
    // Intel Asm syntax
    // jmp label: Is the equivalent of Goto LABEL:

    mov eax, 0      ; set a counter To enumerate the Loop
    mov ebx, num    ; store Var num in ecx register
    top:            ; Label
    cmp eax, ebx    ; Test If eax == num
    je bottom       ; Loop Exit when While condition True
    BODY            ; ... Code inside the Loop
    inc eax         ; Increment the counter +1
    jmp top         ; Jump To label top:
    bottom:         ; Label
    '/

    a = 0 '  reset counter a to 0.
    ' The while loop will print to 4 as the truth test is before the printf().
    ' This is the correct way to implement a loop by using while or for.
    While(1)
        If a = num Then
            Exit While
        End If
        Print a
        a+=1
    Wend

    /'
    ' Assembly Output For the above While loop.
    ' Intel Asm syntax
    .LC0:
    .string "%d,\n"
    main:
    push    rbp
    mov     rbp, rsp
    Sub     rsp, 16
    mov     DWORD Ptr [rbp-8], 5
    mov     DWORD Ptr [rbp-4], 0
    .L4:
    mov     eax, DWORD Ptr [rbp-4]
    cmp     eax, DWORD Ptr [rbp-8]
    je      .L7
    mov     eax, DWORD Ptr [rbp-4]
    mov     esi, eax
    mov     edi, OFFSET FLAT:.LC0
    mov     eax, 0
```

```
    Call    printf
    Add     DWORD Ptr [rbp-4], 1
    jmp     .L4
    .L7:
    nop
    mov     eax, 0
    leave
    ret
    '/

    Con_Pause()

    ' Recreate a while loop (correctly) using goto (jmp)
    ' This is the same as the while loop above.
    ' Start do while loop
    Dim As Integer d = 0
    Goto L2  ' enter while loop.
    L2:
    If d = num Then  ' Truth test
        Goto L7  ' break out of loop
    End If
    Print d
    d+=1  ' Increment d.
    Goto L2
    L7:  ' exit while loop

    /'
    ' Exact assembly output For the  Goto (While) loop.
    ' Intel Asm syntax
    ' eax, esi, etc are CPU registers.
    ' Essentially the internal commands (keywords) or instruction set of the CPU.
    .LC0:
    .string "%d\n"
    main:                               ; main()
    push    rbp                     ; main()set integers
    mov     rbp, rsp                ; main() sets integers
    Sub     rsp, 16                 ; main() sets integers To 16bit
    mov     DWORD Ptr [rbp-8], 5    ; Declare variable num = 5
    mov     DWORD Ptr [rbp-4], 0    ; Declare variable d = 0
    nop
    .L2:                            ; Start While Loop
    mov     eax, DWORD Ptr [rbp-4]
    cmp     eax, DWORD Ptr [rbp-8]  ; Truth test
    je      .L7                     ; If True Exit Loop
    mov     eax, DWORD Ptr [rbp-4]
    mov     esi, eax
    mov     edi, OFFSET FLAT:.LC0
    mov     eax, 0
    Call    printf                  ; Print d
    Add     DWORD Ptr [rbp-4], 1    ; Increment d (+1)
    jmp     .L2                     ; Loop
    .L7:                            ; Exit While Loop
    nop
    mov     eax, 0
    leave                           ; Exit main()
    ret                             ; main() Return statement
    '/
```

```
    Con_Pause()
    Return 0
End Function  ' END main_procedure <---

Function Con_Pause() As Integer
    Dim As Long dummy
    Print("Press any key to continue...")
    dummy = Getkey
    Return 0
End Function
```

Language: Python 3

Code: “Loopsa.py”

```
#-------------------------------------------------------------------------------
# Name:        Loops.py
# Purpose:     Example
#
# Platform:    REPL, Win64, Ubuntu64
#
# Author:      Axle
# Created:     15/02/2022
# Updated:     19/02/2022
# Copyright:   (c) Axle 2022
# Licence:     MIT No Attribution
#-------------------------------------------------------------------------------

def main():

    # A simple example of nested loops using for and while.
    num = 5 #  length of loops.

    # Set variables for enumerating (Counting through) the loops.
    a = 0
    b = 0
    c = 0
    # Loop level 1.
    for a in range(0, num +1):
        print("a" + str(a) + ":")
        # Loop level 2 (nested).
        for b in range(0, a +1):
            print("b" + str(b) + ":", end="")
            # Loop level 3 (nested).
            for c in range(0, b +1):
                print(str(c) + ",", end="")
            print("")
        print("")

    Con_Pause()

    # Reset our counters to zero.
    a = 0
    b = 0
    c = 0
    # Loop level 1.
    while(a <= num):
        print("a" + str(a) + ":")
```

```
        # Loop level 2 (nested).
        while(b <= a):
            print("b" + str(b) + ":", end="")
            # Loop level 3 (nested).
            while(c <= b):
                print(str(c) + ",", end="")
                c+=1
            print("")
            c = 0
            b+=1
        print("")
        b = 0
        a+=1

    Con_Pause()

    # How loops are created in assembly.
    # Intel Asm syntax
    # jmp label: Is the equivalent of Goto LABEL:

    #       mov eax, 0      ; store var num in ecx register
    #       mov ebx, num
    #   top:                ; Label
    #       cmp eax, ebx    ; Test if eax == num
    #       je bottom       ; loop exit when while condition True
    #       BODY            ; ... Code inside the loop
    #       inc eax         ; Increment the counter +1
    #       jmp top         ; Jump to label top:
    #   bottom:             ; Label

    a = 0  #  reset counter a to 0.
    # The while loop will print to 4 as the truth test is before the printf().
    # This is the correct way to implement a loop by using while or for.
    while(1):
        if a == num:
            break
        print(str(a))
        a+=1

    Con_Pause()
    return None

def Con_Pause():
    dummy = ""
    dummy = input("Press [Enter] key to continue...")

    return None

if __name__ == '__main__':
    main()
```

Loops examples part b.

Language: C
Code: “Loopsb.c”

```
//------------------------------------------------------------------------------
// Name:        Loopsb.c
// Purpose:     Loops Animation
//
// Platform:    Win64, Ubuntu64
//
// Author:      Axle
// Created:     16/02/2022
// Updated:     18/02/2022
// Copyright:   (c) Axle 2022
// Licence:     MIT No Attribution
//------------------------------------------------------------------------------
// Check unistd.h Beep(), system("beep")
// https://frank-buss.de/beep/
//------------------------------------------------------------------------------

// C std library headers.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Platform specific headers.
// Test if Windows or Unix OS
#ifdef _WIN32
#   define WIN32_LEAN_AND_MEAN
#   include <Windows.h>
#   define OS_Windows 1  // 1 = True (aka Bool)
#   define OS_Unix 0
#   ifdef _WIN64
#       define __USE_MINGW_ANSI_STDIO 1
#   endif
#endif

#ifdef __unix__ // _linux__
#include <unistd.h>
#define OS_Unix 1
#define OS_Windows 0  // 0 = False (aka Bool)
// Unix requires a complex struct in #def to implement a millisecond sleep
// that is comparable to Win-API Sleep() so I have just changed it to a
// 1 second sleep using Unix sleep(). 1 second is the minimum sleep time
// available for Unix sleep(), so it will be a bit slow on Linux.
// The following line replaces all occurrences of Sleep() with sleep().
// Sleep is a Windows function, and sleep lowers case s is a Unix function.
#define Sleep(x) sleep(1)
#endif

// Turn off compiler warnings for unused variables between (Windows/Linux etc.)
#define unused(x) (x) = (x)

// Define extra functions so we can place them at the bottom of the page.
int Con_Clear(void);
void S_Pause(void);
int S_getchar(void);

int main(int argc, char *argv[])  // Main procedure
    {
    unused(argc);  // Turns off the compiler warning for unused argc, argv
    unused(argv);  // Turns off the compiler warning for unused argc, argv
```

```
    // ---> START "Loops Examples"
    // ---> START "Screen animation one routines"
    // Loading screen animation one.
    // [8] allows room for escape sequences such as end of string '\0'
    // '\' is an escape prefix and must be placed in a literal as '\\'
    static char BarRotate[8] = "|\\-/";
    char Notice[] = "Please wait...";  // Notice[] == *Notice
    float RotSpeed = 200;  //200 This is the sleep timer. Lower setting = faster.
    int counts = 15;  // The number of scenes.
    int x = 0;  // loop counters
    int y = 0;  // loop counters
    int z = 0;  // loop counters

    printf("%s\n", Notice);
    // Keep looping through the 4 characters.
    for(x = 0; x < counts ; x++)
        {
        if(y > 3)  // At the fourth (0,1,2,3) character '/' restart from 0
            {
            y = 0;  // reset to 0 and restart character enumeration
            }
        printf("%c", BarRotate[y]);  // '\b' = Backspace control character.
        fflush(stdout);  // If '\n' not found, push the print to the console.
        Sleep(RotSpeed);  //RotSpeed replaced with cross platform wrapper.
        y++;
        printf("\r");  // Backspace key to clear last character.
        }
    Con_Clear();  // Clear the screen for the next animation.

    // ---> START "Screen animation two routines"
    // Loading screen animation two.
    // Create the array containing the 4 strings.
    // Special characters must be preceded by an escape in side of strings '\'
    static char BigRotate1[4][16] =
        {
            {
            '\\',' ',' ','\n',
            ' ','\\',' ','\n',
            ' ',' ','\\','\0'
            },

            {
            ' ','|',' ','\n',
            ' ','|',' ','\n',
            ' ','|',' ', '\0'
            },

            {
            ' ',' ','/','\n',
            ' ','/',' ','\n',
            '/',' ',' ','\0'
            },

            {
            ' ',' ',' ','\n',
            '-','-','-','\n',
            ' ',' ',' ','\0'
            }
        };
```

```
    // This is the exact same array as above created in string format.
    static char BigRotate2[4][16] =
        {
            {
            "\\  \n"
            " \\ \n"
            "  \\\0"
            },

            {
            " | \n"
            " | \n"
            " | \0"
            },

            {
            "  /\n"
            " / \n"
            "/  \0"
            },

            {
            "   \n"
            "---\n"
            "   \0"
            }
        };

    while(z < counts)  // Repeat counts times.
        {
        for(x = 0; x < 4; x++)  // loop through all 4 array elements.
            {
            printf("%s\n", Notice);
            printf("%s", BigRotate1[x]);  // Alternative use BigRotate2
            fflush(stdout);  // If '\n' not found, push the print to the console.
            Sleep(RotSpeed);  // Slow the animation down.
            Con_Clear();
            }
        z+= 4;
        }

    Con_Clear();  // Clear the screen for the next animation.
    Sleep(500);  // Just a pause to reduce screen/buffer flicker.

    // ---> START "Screen animation three routines"
    // Loading screen animation three.
    static char Tracer[22] = {'\0'};  // Initiate all to 0.
    for(x = 0; x < 21; x++)  // Populate with space, except for string
terminator[22].
        {
        Tracer[x] = ' ';
        }

    int t = 0;  // loop counters
    z = 0;  // Reset loop counters
    int tr_repeats = 0;
    if(1 == OS_Unix)  // Linux
        {
```

```
        tr_repeats = 2;  // for slow Linux sleep(seconds)
        }
    else  // Windows
        {
        tr_repeats = 3;
        }

    while(z < tr_repeats)  // abs(n/5) to reduce the repeats.
        {
        while(t < 20)  // Loop forward
            {
            t++;  // Start at 0 +1 to allow room for the tail
            Tracer[t] = '#';  // Add char at current position.
            Tracer[t-1] = ':';  // Add tail at current position -1.
            printf("%s\n", Notice);
            printf("%s", Tracer);  // Print full array/string to the screen.
            fflush(stdout);  // If '\n' not found, push the print to the console.
            Tracer[t-1] = ' ';  // Remove the tail. Next tail will overwrite #.
            Sleep(RotSpeed/4);  // Slow the animation down a bit.
            Con_Clear();  // Clear the screen for the next print.
            }
        while(t > 0)  // Loop backwards
            {
            t--;  // Start at -1 to allow room for the tail at t.
            Tracer[t] = '#';
            Tracer[t+1] = ':';  // t+1 is the right side of '#'.
            printf("%s\n", Notice);
            printf("%s", Tracer);
            fflush(stdout);
            Tracer[t+1] = ' ';
            Sleep(RotSpeed/4);
            Con_Clear();
            }
        Sleep(RotSpeed/4);
        z+= 1;
        }

    Con_Clear();  // Clear the screen for the next animation.
    Sleep(500);  // Just a pause to reduce screen/buffer flicker.

    // ---> START "Stick-man animation routines"
    // Stick-man animation
    // Define our data arrays...
    // This is the original ASCII text file from the net.
    // The arrays were created by hand in Notepad++ on this occasion.
    //  o   \ o /  _ o        __|    \ /     |__        o _  \ o /   o     |
    // /|\    |     /\   ___\o   \o    |    o/    o/__   /\     |    /|\    |
    // / \   / \   | \  /)  |    ( \  /o\  / )    |  (\  / |   / \   / \    |

    int Len_Cart = 11;  // Keep a record of the array length.
    // The second element is the size of the string buffer...
    // Note: We must use an escape character for special characters \\ = '\'.
    static char Cart1[11][60] =  // [11][17*3]
        {
            {
            ' ',' ',' ','o',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','\n',
            ' ',' ','/','|','\\',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','\n',
            ' ',' ','/',' ','\\',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','\n','\0'
            },
```

```
{
' ','\\',' ','o',' ','/',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','\n',
' ',' ',' ','|',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','\n',
' ',' ','/',' ','\\',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','\n','\0'
},

{
' ',' ','_',' ','o',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','\n',
' ',' ',' ','/',' ','\\',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','\n',
' ',' ','|',' ','\\',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','\n','\0'
},

{
' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','\n',
' ',' ',' ','_','_','_','\\','o',' ',' ',' ',' ',' ',' ',' ','\n',
' ',' ','|',')',' ',' ','|',' ',' ',' ',' ',' ',' ',' ',' ','\n','\0'
},

{
' ',' ',' ',' ','_','_','|',' ',' ',' ',' ',' ',' ',' ',' ',' ','\n',
' ',' ',' ',' ',' ',' ','\\','o',' ',' ',' ',' ',' ',' ',' ','\n',
' ',' ',' ',' ',' ',' ','(',' ','\\',' ',' ',' ',' ',' ',' ','\n','\0'
},

{
' ',' ',' ',' ',' ',' ','\\',' ','/',' ',' ',' ',' ',' ',' ','\n',
' ',' ',' ',' ',' ',' ',' ','|',' ',' ',' ',' ',' ',' ',' ','\n',
' ',' ',' ',' ',' ',' ','/','o','\\',' ',' ',' ',' ',' ',' ','\n','\0'
},

{
' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','|','_','_',' ',' ',' ','\n',
' ',' ',' ',' ',' ',' ',' ','o','/',' ',' ',' ',' ',' ',' ','\n',
' ',' ',' ',' ',' ',' ','/',' ',')',' ',' ',' ',' ',' ',' ','\n','\0'
},

{
' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','\n',
' ',' ',' ',' ',' ',' ',' ',' ',' ','o','/',' ','_',' ',' ',' ','\n',
' ',' ',' ',' ',' ',' ',' ',' ',' ','|',' ',' ','(','\\',' ',' ','\n','\0'
},

{
' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','o',' ',' ','_',' ','\n',
' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','/','\\',' ',' ','\n',
' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','/',' ',' ','|',' ','\n','\0'
},

{
' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','\\',' ','o',' ',' ','/','\n',
' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','|',' ',' ',' ','\n',
' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','/',' ',' ','\\',' ','\n','\0'
},

{
' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','o',' ',' ',' ','\n',
' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','/','|','\\',' ',' ','\n',
' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','/',' ',' ','\\',' ','\n','\0'
```

```
            }
        };

    // This is the exact same array as above created in string format.
    // Note: We must use an escape character for special characters \\ = '\'.
    static char Cart2[11][60] =  // [11][18*3] <- 60 just to be safe.
        {
            {
            "   o           \n"
            "  /|\\           \n"
            "  / \\           \n\0"
            },

            {
            " \\ o /          \n"
            "   |            \n"
            "  / \\           \n\0"
            },

            {
            "  _ o           \n"
            "   /\\           \n"
            "  | \\           \n\0"
            },

            {
            "                \n"
            "   ___\\o        \n"
            "  |)  |          \n\0"
            },

            {
            "    __|         \n"
            "      \\o        \n"
            "      ( \\       \n\0"
            },

            {
            "      \\ /       \n"
            "       |        \n"
            "      /o\\       \n\0"
            },

            {
            "         |__    \n"
            "       o/       \n"
            "      / )       \n\0"
            },

            {
            "                \n"
            "        o/__    \n"
            "        |  (\\   \n\0"
            },

            {
            "           o _  \n"
            "           /\\   \n"
            "           / |  \n\0"
```

```
            },

            {
            "           \\ o /\n"
            "             |  \n"
            "            / \\ \n\0"
            },

            {
            "              o  \n"
            "             /|\\ \n"
            "             / \\ \n\0"
            }
        };

    int a = 0;  // Loop counters
    int b = 0;  // Loop counters
    int Speed = 400;  // Delay in milliseconds to slow the animation down.
    unused(Speed);  // Turns off the compiler warning when in Linux
    int C_repeat = 0;  // Repeat the animation for loops 3 times.
    if(1 == OS_Unix)  // Linux
        {
        C_repeat = 2;  // For slow Linux sleep(seconds)
        }
    else  // Windows
        {
        C_repeat = 3;
        }

    while(a < C_repeat)  // Repeat the animation n times.
        {
        for(b = 0; b < Len_Cart; b++)  // Loop animation forward
            {
            Con_Clear();  // Clear the console ready for the next print.
            printf("%s", Cart1[b]);  // Alternative use Cart2
            fflush(stdout);  // If '\n' not found, push the print to the console.
            Sleep(Speed);  // replace with cross platform wrapper.
            }
        for(b = Len_Cart -1; b >= 0; b--)  // Loop animation backwards
            {
            Con_Clear();  // Clear the console ready for the next print.
            printf("%s", Cart1[b]);  // Alternative use Cart2
            fflush(stdout);
            Sleep(Speed);  // replaced with cross platform wrapper.
            }
        a++;
        }

    // END Loops examples <---

    S_Pause();  // DEBUG Pause
    return 0;
    }  // END main() <---

// --> START helper functions

// Console Clear
int Con_Clear(void)
    {
```

```
    // The system() call allows the programmer to run OS CLI batch commands.
    // It is discouraged as there are more appropriate C functions for most tasks.
    // I am only using it in this instance to avoid invoking additional OS API
    // headers and code.
    if(OS_Windows)
        {
        system("cls");
        }
    else if(OS_Unix)
        {
        system("clear");
        }
    return 0;
    }

// Safe Pause
void S_Pause(void)
    {
    // This function is referred to as a wrapper for S_getchar()
    printf("Press any key to continue...");
    S_getchar();  // Uses S_getchar() for safety.
    }

// Safe getcar() removes all artefacts from the stdin buffer.
int S_getchar(void)
    {
    // This function is referred to as a wrapper for getchar()
    int i = 0;
    int ret;
    int ch;
    // The following enumerates all characters in the buffer.
    while((ch = getchar()) != '\n' && ch != EOF )
        {
        // But only keeps and returns the first char.
        if (i < 1)
            {
            ret = ch;
            }
        i++;
        }
    return ret;
    }
```

Language: FreeBASIC

Code: “Loopsb.bas”

```
'--------------------------------------------------------------------------------
' Name:         Loopsb.bas
' Purpose:      Loop animation
'
' Platform:     Win64, Ubuntu64
'
' Author:       Axle
' Created:      16/02/2022
' Updated:      19/02/2022
' Copyright:    (c) Axle 2022
' Licence:      MIT No Attribution
```

```
'--------------------------------------------------------------------------------

' Test if Windows or Unix OS
#ifdef __FB_WIN32__
#define OS_Windows 1  ' 1 = True (aka Bool)
#define OS_Unix 0
#endif

#ifdef __FB_UNIX__'__FB_LINUX__
' TODO
#define OS_Unix 1
#define OS_Windows 0  ' 0 = False (aka Bool)
#endif

' Define extra functions so we can place them at the bottom of the page.

Declare Function Con_Clear() As Integer
Declare Function Con_Pause() As Integer  ' GetKey Version
Declare Function main_procedure() As Integer

main_procedure()

Function main_procedure() As Integer  ' Main procedure

    ' Loading screen animation one.
    ' '\' is an escape prefix and must be placed in a literal as '\\'
    Dim As String BarRotate = "|\-/"
    Dim As String Notice = "Please wait..."
    Dim As Integer RotSpeed = 500  ' This is the sleep timer. Lower setting =
faster.
    Dim As Integer counts = 15  ' The number of scenes.
    Dim As Integer x = 0  ' loop counters
    Dim As Integer y = 0  ' loop counters
    Dim As Integer z = 0  ' loop counters

    Print Notice
    ' Keep looping through the 4 characters.
    For x = 0 To counts -1 Step 1
        If(y > 3) Then  ' At the fourth (0,1,2,3) character '/' restart from 0
            y = 0  ' reset to 0 and restart character enumeration.
        End If
        Print Chr(BarRotate[y]);  ' '\b' = Backspace control character.
        Sleep(RotSpeed)  ' replaced with cross platform wrapper.
        y+= 1
        Print !"\r";  ' Carriage Return to overwrite the last character.
    Next x

    Con_Clear()
    Sleep(500)  ' Just a pause to reduce screen/buffer flicker.

    ' Loading screen animation two.
    ' Create the array containing the 4 strings.
    ' Special characters must be preceded by an escape in side of strings '\'
    ' ' _' is a line continuation so we can break a long line over several.
    Dim As String BigRotate2(4) => { _
    !"\\  \n" _
    !" \\ \n" _
    !"  \\", _
    !" | \n" _
```

```
    !" | \n" _
    " | ", _
    !"  /\n" _
    !" / \n" _
    "/  ", _
    !"   \n" _
    !"---\n" _
    "   "}

    While(z < counts)  ' Repeat counts times.
        For x = 0 To 4 -1 Step 1  ' loop through all 4 array elements.
            Print Notice
            Print BigRotate2(x);  ' Alternative use BigRotate2
            Sleep(Abs(RotSpeed/2))  ' Slow the animation down.
            Con_Clear()
        Next x
        z+= 4
    Wend

    Con_Clear()  ' Clear the screen for the next animation.
    Sleep(500)  ' Just a pause to reduce screen/buffer flicker.

    ' Loading screen animation 3.
    Dim As String Tracer = ""  ' Initiate all to 0.
    For x = 0 To 21 -1 Step 1  ' Populate with space, except for string
terminator[22].
        Tracer+= " "
    Next

    Dim As Integer t = 0  ' loop counters
    z = 0  ' Reset loop counters
    While(z < (Abs(counts/5)))  ' abs(n/5) to reduce the repeats.
        While(t < 20)  ' Loop forward...
            t += 1  ' Start at 0 +1 to allow room for the tail.
            Tracer[t] = Asc("#")  ' Add mark at current position.
            Tracer[t-1] = Asc(":")  ' Add tail at current position -1.
            Print Notice
            Print Tracer;  ' Print full array/string to the screen.
            Tracer[t-1] = Asc(" ")  ' Remove the tail. Next tail (+1) will
overwrite the Marker.
            Sleep(RotSpeed/4)  ' slow the animation down a bit
            Con_Clear()  ' Clear the screen for the next print.
        Wend
        While(t > 0)  ' Loop backwards...
            t -= 1  ' Start at -1 to allow room for the tail at t.
            Tracer[t] = Asc("#")
            Tracer[t+1] = Asc(":")  ' t+1 is the right side of '#'.
            Print Notice
            Print Tracer;
            Tracer[t+1] = Asc(" ")
            Sleep(RotSpeed/4)
            Con_Clear()
        Wend
        z+= 1
    Wend

    Con_Clear()  ' Clear the screen for the next animation.
    Sleep(500)  ' Just a pause to reduce screen/buffer flicker.
```

```
    ' Stick-man animation
    ' Define our data arrays...
    ' This is the original ASCII text file from the net.
    ' The arrays were created by hand in Notepad++ on this occasion.
    '  o   \ o /  _ o        __|    \ /     |__        o _  \ o /   o
    ' /|\    |     /\   ___\o   \o    |    o/    o/__   /\     |    /|\
    ' / \   / \   | \  /)  |    ( \  /o\  / )    |  (\  / |   / \   / \

    Dim As Integer Len_Cart = 11  ' Keep a record of the array length.
    ' Note: the '!' before a string literal allows the use of escape chars '\'
    ' ' _' is a line continuation so we can break a long line over several.
    Dim As String Cart2(11) => { _
    !"   o           \n" _
    !"  /|\\          \n" _
    !"  / \\          ", _
    !" \\ o /         \n" _
    !"   |           \n" _
    !"  / \\          ", _
    !"  _ o          \n" _
    !"   /\\          \n" _
    !"  | \\          ", _
    !"               \n" _
    !"   ___\\o       \n" _
    !"  |)  |        ", _
    !"    __|        \n" _
    !"      \\o       \n" _
    !"      ( \\      ", _
    !"      \\ /      \n" _
    !"       |       \n" _
    !"      /o\\      ", _
    !"         |__   \n" _
    !"       o/      \n" _
    !"      / )      ", _
    !"               \n" _
    !"        o/__   \n" _
    !"        |  (\\  ", _
    !"           o _ \n" _
    !"           /\\  \n" _
    !"           / | ", _
    !"          \\ o /\n" _
    !"            |  \n" _
    !"           / \\ ", _
    !"            o  \n" _
    !"           /|\\ \n" _
    !"           / \\ "}

    Dim As Integer a = 0  ' loop counters
    Dim As Integer b = 0  ' loop counters
    Dim As Integer repeat = 3  ' Repeat the animation for loops 3 times.
    Dim As Integer Speed = 400  ' Delay in milliseconds to slow the animation
down.

    While(a < repeat)  ' Repeat the animation n times.
        For b = 0 To Len_Cart -1 Step 1  ' Loop animation forward
            Con_Clear()  ' Clear the console ready for the next print.
            Print Cart2(b)  ' Print array element [n]
            Sleep(Speed)  ' Replace with cross platform wrapper.
        Next b
        For b = Len_Cart -1 To 0 Step -1  ' Loop animation backwards
```

```
            Con_Clear()  ' Clear the console ready for the next print.
            Print Cart2(b)  ' Print array element [n]
            Sleep(Speed)  ' Replace with cross platform wrapper.
        Next b
        a+= 1
        Sleep(Speed)
    Wend

    Con_Pause() ' DEBUG Pause
    Return 0
End Function  ' END main_procedure <---


' --> START helper functions

' Console Clear
Function Con_Clear() As Integer
    ' The system() call allows the programmer to run OS command line batch
commands.
    ' It is discouraged as there are more appropriate C functions for most tasks.
    ' I am only using it in this instance to avoid invoking additional OS API
headers and code.
    If (OS_Windows) Then
        Shell "cls"
    Elseif (OS_Unix) Then
        Shell "clear"
    End If
    Return 0
End Function

' Console Pause (GetKey version)
Function Con_Pause() As Integer
    Dim As Long dummy
    Print("Press any key to continue...")
    dummy = Getkey
    Return 0
End Function
```

Language: Python 3

Code: “Loopsb.py”

```
#-------------------------------------------------------------------------------
# Name:        Loopsb.py
# Purpose:     Loops Animations
#
# Platform:    REPL*, Win64, Ubuntu64
#
# Author:      Axle
# Created:     16/02/2022
# Updated:     19/02/2022
# Copyright:   (c) Axle 2022
# Licence:     MIT No Attribution
#-------------------------------------------------------------------------------
# * This example is best run in a native OS console window.
#-------------------------------------------------------------------------------

def main():
```

```
    if 1 == Con_IsREPL():
        print("This application is best viewed running from the OS Command
interpreter")
        Con_Pause()

    # Loading screen animation one.
    BarRotate = "|\-/"
    Notice = "Please wait..."
    RotSpeed = 0.5  # This is the sleep timer in seconds. Lower setting = faster.
    counts = 15  # The number of scenes.
    x = 0  # loop counters
    y = 0  # loop counters
    z = 0  # loop counters

    print(Notice)
    # Keep looping through the 4 characters in BarRotate.
    for x in range(0, counts):
        if(y > 3): # At the forth (0,1,2,3) character '/' restart from 0
            y = 0  # reset to 0 and restart character enumeration.
        # Python will hold the print statements until a new line is encountered.
        # Use flush to push the print character to the console stdout.
        print(BarRotate[y], end='', flush=True)
        Con_sleep(RotSpeed)  # Sleep(pause) for a moment to slow the animation.
        y+= 1
        # I was using backspace '\b', but it doesn't work on all consoles or REPL.
        print("\r", end="", flush=True)  # '\r' = Carriage return to overwrite the
last character.

    Con_Clear()  # Clear the console stdout screen.
    Con_sleep(0.5)  # Just a pause to reduce screen/buffer flicker.

    # Loading screen animation two.
    # Create the list containing the 4 strings.
    # Special characters must be preceded by an escape inside of strings '\'
    BigRotate2 = [
    "\\  \n"
    " \\ \n"
    "  \\",
    " | \n"
    " | \n"
    " | ",
    "  /\n"
    " / \n"
    "/  ",
    "   \n"
    "---\n"
    "   "]

    while(z < counts):  # Repeat counts times.
        for x in range(0, 4):  # loop through all 4 list elements.
            print(Notice)
            print(BigRotate2[x], end="", flush=True)
            Con_sleep(RotSpeed/2)  # Slow the animation down.
            Con_Clear()
        z+= 4
    print("")  # Reset carret (Cursor) CRLF for next print statements.

    Con_Clear()  # Clear the screen for the next animation.
```

```
    Con_sleep(0.5)  # Just a pause to reduce screen/buffer flicker.

    # Loading screen animation 3.
    Tracer = ""  # Initiate all to 0.
    for x in range(0, 21):  # Populate with space, except for string
terminator[22]
        Tracer+= " "

    t = 0  # loop counters
    z = 0  # Reset loop counters
    while(z < (abs(counts/4))):  # abs(n/5) to reduce the repeats.
        while(t < 20):  # Loop forward...
            t += 1  # Start at 0 +1 to allow room for the tail.
            # !!! It may be more efficient to run this from a list rather
            # than rebuild the string "Tracer" 3 times in each loop.
            Tracer = Tracer[:t] + "#" + Tracer[t+1:]  # Rebuild the string with
            Tracer = Tracer[:t-1] + ":" + Tracer[t:]  # the new characters in
place.
            print(Notice)
            print(Tracer, end="", flush=True)  # Print full array/string to the
screen.
            Tracer = Tracer[:t-1] + " " + Tracer[t:]  # clear a character from the
string.
            Con_sleep(RotSpeed/10)  # slow the animation down a bit
            if 1 == Con_IsREPL():  # REPL vs OS Console.
                print("")
            Con_Clear()  # Clear the screen for the next print.
        while(t > 0):  # Loop backwards...
            t -= 1  # Start at -1 to allow room for the tail at t.
            Tracer = Tracer[:t] + "#" + Tracer[t+1:]
            Tracer = Tracer[:t+1] + ":" + Tracer[t+2:]  # t+1 is the right side of
'#'.
            print(Notice)
            print(Tracer, end="", flush=True)
            Tracer = Tracer[:t+1] + " " + Tracer[t+2:]
            Con_sleep(RotSpeed/10)
            if 1 == Con_IsREPL():
                print("")
            Con_Clear()
        z+= 1
    print("")  # Reset carret (Cursor) CRLF for next print statements.

    Con_Clear()  # Clear the screen for the next animation.
    Con_sleep(0.5)  # Just a pause to reduce screen/buffer flicker.

    # Stick-man animation
    # Define our data arrays...
    # This is the original ASCII text file from the net.
    # The arrays were created by hand in Notepad++ on this occasion.
     # o   \ o /  _ o        __|    \ /     |__        o _  \ o /   o
    # /|\    |     /\   ___\o   \o    |    o/    o/__   /\     |    /|\
    # / \   / \   | \  /)  |    ( \  /o\  / )    |  (\  / |   / \   / \

    Len_Cart = 11  # Keep a record of the list length.
    # Note: We must use an escape character for special characters \\ = '\'.
    Cart2 = [
    "   o           \n"
    "  /|\\          \n"
    "  / \\          ",
```

```
    " \\ o /          \n"
    "   |            \n"
    "  / \\           ",
    "  _ o           \n"
    "   /\\           \n"
    "  | \\           ",
    "                \n"
    "   ___\\o        \n"
    "  |)  |         ",
    "    __|         \n"
    "      \\o        \n"
    "      ( \\       ",
    "      \\ /       \n"
    "       |        \n"
    "      /o\\       ",
    "         |__    \n"
    "       o/       \n"
    "      / )       ",
    "                \n"
    "        o/__    \n"
    "        |  (\\   ",
    "           o _  \n"
    "           /\\   \n"
    "           / |  ",
    "          \\ o /\n"
    "            |   \n"
    "           / \\  ",
    "            o   \n"
    "           /|\\  \n"
    "           / \\  "]

    a = 0  # loop counters
    b = 0  # loop counters
    repeat = 3  # Repeat the animation for loops 3 times.
    Speed = 0.3  # Delay in seconds to slow the animation down.

    while(a < repeat):  # Repeat the animation n times.
        for b in range(0, Len_Cart):  # Loop animation forward.
            Con_Clear()  # Clear the console ready for the next print.
            print(Cart2[b])  # Print list element [n]
            Con_sleep(Speed)  # Replace with cross platform wrapper.
        Con_sleep(Speed)  # slow the animation down a bit.
        for b in range(Len_Cart-1, -1, -1):  # Loop animation backwards.
            Con_Clear()  # Clear the console ready for the next print.
            print(Cart2[b])  # Print list element [n]
            Con_sleep(Speed)  # slow the animation down a bit.
        a+= 1
        Con_sleep(Speed)  # To reduce screen flicker (Visual effect)

    Con_Pause()  # DEBUG Pause
    return None
    # END Main() <---


# --> START helper functions

# Test if we are inside of the REPL interactive interpreter.
# This function is in alpha and may not work as expected.
def Con_IsREPL():
```

```
    import os
    if os.sys.stdin and os.sys.stdin.isatty():
        if os.isatty(os.sys.stdout.fileno()):
            return 0  # OS Command Line
        else:
            return 1  # REPL - Interactive Linux?
    else:
        return 1  # REPL - Interactive Windows?
    return None

# Cross platform console clear.
# This function is in alpha and may not work as expected.
def Con_Clear():
    # The system() call allows the programmer to run OS command line batch
commands.
    # It is discouraged as there are more appropriate C functions for most tasks.
    # I am only using it in this instance to avoid invoking additional OS API
headers and code.
    import os
    if os.sys.stdin and os.sys.stdin.isatty():
        if os.isatty(os.sys.stdout.fileno()):# Clear function doesn't work in Py
REPL
            # for windows
            if os.name == 'nt':
                os.system('cls')
            # for mac and linux
            elif os.name == 'posix':
                os.system('clear')
            else:
                return None  # Other OS
        else:
            return None  # REPL - Interactive Linux?
    else:
    return None  # REPL - Interactive Windows?

# Console Pause wrapper.
def Con_Pause():
    dummy = ""
    dummy = input("Press [Enter] key to continue...")
    return None

# Console sleep
def Con_sleep(times: float):
    import time
    time.sleep(times)
    return None

if __name__ == '__main__':
    main()
```

Functions

Reusable Blocks of code, Function name, Gosub, Subroutine

Functions are reusable “Blocks” of code. Write it once and re-use it as many times as needed, often billions of re-uses per second!

Functions allow us to create “Modular” code that we can create and then reuse many times within an application without having to write duplications of the same routine over and over. Functions are also transportable between different projects.

Over time we may collect a significant number of preferred functions that we use regularly in different applications. As we have created the solution in a code routine, and placed it in a function we don’t have to repeat the hard work of writing it the first time. This “Collection” of functions will become a personal “Library” and in many cases they will be placed in a separate “Include” file that we can call at the top of our source code page.

#include “.\Helper_Functs.h”
#include Once .\Helper_Functs.bi"
import “.\Helper_Functs.py”

This process of creating a collection of functions is the basis that specialized libraries are created upon. An Audio library such as Bass Audio for example is a collection of functions that make it easy for the programmer to interact with the operating system’s audio hardware.

Language: C

Code: “Functions.c”

```
//------------------------------------------------------------------------------
// Name:        Functions.c
// Purpose:     Examples
//
// Platform:    Win64, Ubuntu64
//
// Author:      Axle
// Created:     21/02/2022
// Updated:     22/02/2022
// Copyright:   (c) Axle 2022
// Licence:     MIT No Attribution
//------------------------------------------------------------------------------


// C std library headers.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

// Platform specific headers.
// Test if Windows or Unix OS
#ifdef _WIN32
#   define WIN32_LEAN_AND_MEAN
#   include <Windows.h>
#   define OS_Windows 1  // 1 = True (aka Bool)
#   define OS_Unix 0
#   ifdef _WIN64
#       define __USE_MINGW_ANSI_STDIO 1
#   endif
#endif
```

```
#ifdef __unix__ // _linux__
#include <unistd.h>
#define OS_Unix 1
#define OS_Windows 0  // 0 = False (aka Bool)
// Unix requires a complex struct in #def to implement a millisecond sleep
// that is comparable to Win-API Sleep() so I have just changed it to a
// 1 second sleep using Unix sleep(). 1 second is the minimum sleep time
// available for Unix sleep(), so it will be a bit slow on Linux.
// The following line replaces all occurrences of Sleep() with sleep().
// Sleep is a Windows function, and sleep lowers case s is a Unix function.
#define Sleep(x) sleep(1)
#endif

// Turn off compiler warnings for unused variables between (Windows/Linux etc.)
#define unused(x) (x) = (x)


// Define extra functions so we can place them at the bottom of the page.
int Menu_Routine(void);
int Stick_Animation(void);
int File_Write_Example(void);
int File_Read_Example1(void);
int File_Read_Example2(void);
int Mandelbrot_Fractals_Console_ASCII(void);


int Sys_Sound(void);
int DebugMsg(char *aTitle, char *aMessage);
int Con_Clear(void);
void S_Pause(void);
int S_getchar(void);
// v Emulates Input function in FreeBASIC and Python 3.
char *Input(char *buf, char *str, int n);
char *S_fgets(char *buf, int n, FILE *stream);
int Show_Time_Date(void);
int show_time(void);
int show_date(void);

int main(int argc, char *argv[])  // Main procedure
    {
    unused(argc);  // Turns off the compiler warning for unused argc, argv
    unused(argv);  // Turns off the compiler warning for unused argc, argv

    Menu_Routine();

    S_Pause();  // DEBUG Pause
    return 0;
    }  // END main() <---


// --> START Application Functions
int Menu_Routine(void)
    {
    while(1)
        {
        // Forever loop. Needs a break, return, or exit() statement to
        // exit the loop.
        Con_Clear();
        // Placing the menu inside of the while loop "Refreshes" the menu
```

```
        // if an incorrect value is supplied.
        char inputbuffer[4] = {'\0'};  // Inputs are as string.
        int option = 0;  // Menu variable.

        printf("============================================\n");
        printf("  MAIN MENU\n");
        printf("============================================\n");
        printf("  1 - Stick Animation\n");
        printf("  2 - File Write Example\n");
        printf("  3 - File Read Example 1\n");
        printf("  4 - File Read Example 2\n");
        printf("  5 - Debug Msg Box\n");
        printf("  6 - System Sound Test\n");
        printf("  7 - Mandelbrot Fractals\n");
        printf("  8 - Display time and date\n");
        printf("\n");
        printf("  9 - Exit The Application\n");
        printf("--------------------------------------------\n");
        //printf("Enter your menu choice: ");
        // atoi() Convert char/Str to an integer.
        option = atoi(Input(inputbuffer, "  Enter your menu choice: ", 2));
        printf("\n");

        // Check what choice was entered and act accordingly
        // We can add as many choices as needed
        if(option == 0)
            {
            // Ignore false [Enter]s
            }
        else if(option == 1)
            {
            Stick_Animation();
            }
        else if(option == 2)
            {
            File_Write_Example();
            }
        else if(option == 3)
            {
            File_Read_Example1();
            }
        else if(option == 4)
            {
            File_Read_Example2();
            }
        else if(option == 5)
            {
            // Send String. "Hello"
            char aVariable[] = "This is my message.";
            DebugMsg("DEBUGmsg", aVariable);
            // Send  Int 125
            //int iVariable = 125;
            //char Buffer[128] = {'\0'};
            //DebugMsg("DEBUGmsg", itoa(iVariable, Buffer, 10));
            // Alternative conversion atoi(), ftoa(), sprintf().
            }
        else if(option == 6)
            {
            Sys_Sound();
```

```
            }
        else if(option == 7)
            {
            Mandelbrot_Fractals_Console_ASCII();
            }
        else if(option == 8)
            {
            Show_Time_Date();
            S_Pause();
            Con_Clear();
            }
        else if(option == 9)
            {
            printf("Exiting the application...\n");
            Sleep(1000);  // Allow 1 second for the Exit notice to display.
            break;
            }
        else
            {
            Sys_Sound();
            printf("Invalid option.\nPlease enter a number from the Menu
Options.\n\n");
            Sleep(1000);
            }
        }

    return 0;
    }

int Stick_Animation(void)
    {
    // Stick man animation
    int Len_Animation = 11;  // Keep a record of the array length.
    static char Animation[11][60] =
        {
            {
            "   o           \n"
            "  /|\\           \n"
            "  / \\           \n\0"
            },

            {
            " \\ o /         \n"
            "   |           \n"
            "  / \\           \n\0"
            },

            {
            "  _ o          \n"
            "   /\\           \n"
            "  | \\           \n\0"
            },

            {
            "               \n"
            "   ___\\o        \n"
            "  |)  |         \n\0"
            },
```

```
            {
            "     __|          \n"
            "       \\o         \n"
            "       ( \\        \n\0"
            },

            {
            "       \\ /        \n"
            "        |         \n"
            "       /o\\        \n\0"
            },

            {
            "          |__     \n"
            "        o/        \n"
            "       / )        \n\0"
            },

            {
            "                  \n"
            "         o/__     \n"
            "         |  (\\    \n\0"
            },

            {
            "             o _  \n"
            "             /\\   \n"
            "             / |  \n\0"
            },

            {
            "           \\ o /\n"
            "             |   \n"
            "             / \\ \n\0"
            },

            {
            "              o   \n"
            "             /|\\  \n"
            "             / \\  \n\0"
            }
        };

    int a = 0;
    int b = 0;
    int Speed = 400;  // Delay in milliseconds to slow the animation down.
    unused(Speed);  // Turns off the compiler warning when in Linux.

    // Adjust different Sleep function between Windows Unix.
    int C_repeat = 0;
    if(1 == OS_Unix)  // Linux
        {
        C_repeat = 2;  // For slow Linux sleep(seconds)
        }
    else  // Windows
        {
        C_repeat = 3;
        }
```

```
    // Display the animation.
    while(a < C_repeat)
        {
        for(b = 0; b < Len_Animation; b++)  // Loop animation forward
            {
            Con_Clear();
            printf("%s", Animation[b]);
            fflush(stdout);
            Sleep(Speed);
            }
        for(b = Len_Animation -1; b >= 0; b--)  // Loop animation backwards
            {
            Con_Clear();
            printf("%s", Animation[b]);
            fflush(stdout);
            Sleep(Speed);
            }
        a++;
        }
    Con_Clear();
    return 0;
    }

int File_Write_Example(void)
    {
    FILE * fpFileOut;  // File open handle
    // Output file to write to.
    char filename[] = "MyTextFile.txt";

    const int Max_str_Length = 128;
    char string_Temp_Buffer[128] = {'\0'};
    printf("Please enter the text you would like to write to file %s\n",
filename);
    Input(string_Temp_Buffer, "Type a line of text ... followed by [Enter]\n",
Max_str_Length);

    //==> Open Output file for text append ops.
    // "a+" will create a new file '+' if it does not exist.
    fpFileOut = fopen(filename, "a+");
    if(fpFileOut == NULL)  // (!fpFileOut) alt. Test if file open success.
        {
        printf("ERROR! Cannot open Output file %s\n", filename);
        S_Pause();
        return -1;
        }

    fprintf(fpFileOut, "%s\n", string_Temp_Buffer);  // write the buffer to file.

    // we must always remember to close the file when finished.
    fclose(fpFileOut);

    printf("\nFile write completed...\n");
    S_Pause();
    Con_Clear();
    return 0;
    }

// A simple routine that reads a files contents directly to the screen
// using a single buffers.
```

```
int File_Read_Example1(void)
    {
    FILE *fpFileIn;  // File open handle
    char filename[] = "MyTextFile.txt";  // Output file.
    char buffer[128] = {'\0'};
    int len_buffer = 128 -4;  // allow space for '\r','\n','\0'
    int cnt1;  // Loop counters.

    fpFileIn = fopen(filename, "r");  // Open file for read ops
    if(fpFileIn == NULL)  //(!fpFileIn) alt. Test if file open success.
        {
        printf("Error in opening Data file : %s\n", filename);
        printf("Maybe the file has not yet been created.\n");
        printf("Please select from the MAIN Menu\n");
        printf("to create a new file.\n");
        S_Pause();
        return 0;
        }

    while(fgets(buffer, len_buffer, fpFileIn) != NULL)
        {
        // Walk each line from the file (returns the string
        // with '\n' at the end).
        // Strip the newline character from the line and
        // replace newline char '\n' '\r' with '\0'.
        //string_Temp_Buffer[strcspn(string_Temp_Buffer, "\r\n")] = '\0';
        // Copy the cleaned line of text into our array.
        printf("%s", buffer);
        cnt1++;  // move to the next line and repeat.
        }

    // It is important to free up resources as soon as they are no longer
required.
    fclose(fpFileIn);  // finished file reads, close the file.

    S_Pause();
    Con_Clear();
    return 0;
    }

// File read example that copies the file contents into a "Dynamic" array so
// that it can be manipulated. Dynamic arrays are created on the heap.
int File_Read_Example2(void)
    {
    FILE *fpFileIn;  // File open handle
    char filename[] = "MyTextFile.txt";

    // In C we also have to allocate enough char space to hold the string values
    // to be stored. Because we don't know in advance how large the text file
    // will be, we also have to create a dynamic array in memory "On the heap"
    // at run time.
    // An alternative method is to read the contents from the file directly to
    // the screen without the need to test lengths or create a buffer, but I
    // wanted to show how to actually store the data in the application so that
    // it can be manipulated after being read. You could then write the modified
    // text back to the file or to another file; for example "File Copy".

    int Char_Buffer;  // Buffer to hold each character.
    int Total_Lines = 0;  // = Total lines in the text file.
```

```
    int Line_Width = 0;  // Count line widths.
    int Max_Width = 0;  // Total width of the longest line.
    int cnt1, cnt2;  // Loop counters.

    // It is possible that the file may not yet exist. Opening it
    // as "r" will return an exception. Let's test if the file exists first.
    fpFileIn = fopen(filename, "r");  // Open file for read ops
    if(fpFileIn == NULL)  //(!fpFileIn) alt. Test if file open success.
        {
        printf("Error in opening Data file : %s\n", filename);
        printf("Maybe the file has not yet been created.\n");
        printf("Please select from the MAIN Menu\n");
        printf("to create a new file.\n");
        S_Pause();
        return 0;
        }
    else  // Continue to process text file...
        {
        // For obtaining a character count to build our dynamic array.
        fseek(fpFileIn, 0, SEEK_END);  // Set pointer to end of file.
        int char_Total = ftell(fpFileIn);  // get counter value.
        rewind(fpFileIn);  // Set pointer back to the start of the file.

        // Read Character by character and check for new line '\n'.
        // I am testing every char in the file rather than testing line by line.
        for(cnt1 = 0; cnt1 < char_Total; cnt1++)
            {
            Char_Buffer = fgetc(fpFileIn);

            if(Char_Buffer == '\n')  // Test if we have encountered a new line
and,
                {
                Total_Lines++;  // increment the number of new lines in the file.
                if(Line_Width > Max_Width)
                    {
                    // Find and store the longest line.
                    Max_Width = Line_Width;
                    // Reset the width counter for the next line.
                    Line_Width = 0;
                    }
                }
            Line_Width++;
            }
        // Set pointer to start of file.
        // (Start the next file read from the first character of first line.)
        rewind(fpFileIn);

        // Create a temp buffer with Max_Width size to hold each line.
        char *string_Temp_Buffer;
        if((string_Temp_Buffer = malloc((Max_Width +4) * sizeof(char))) != NULL)
            {
            for(cnt1=0; cnt1 < Max_Width +4; cnt1++)
                {
                string_Temp_Buffer[cnt1] = '\0';  // Initialise the array to nul
                }
            }
        else
            {
            // This constitutes an application failure from which we must close
```

```
            // the application. The user should never receive this error! :)
            printf("Error - unable to allocate required memory for temp
buffer.\n");
            S_Pause();
            return -1;
            }

        // Now that we know how many lines (Total_Lines) to allocate, and the
        // length of the longest line (Max_Width) we can create a suitable
        // sized array to hold the contents.

        // char Read_Buffer[lines/Total_Lines][Max_Width]
        // NOTE! The extra + 4 characters is required to hold additional control
        // characters beyond the length of the string. The string "Hello/n" is 5
        // characters long. This is stored as 'H','e','l','l','o','\r','\n','\0'
        // The '\r','\n','\0' requires an extra 3 characters.
        // I just add 4 for safety :)
        // I add an extra +1 lines for reading files for a safety buffer.
        // This is somewhat advanced and beyond the scope of a beginner, but I
        // have no other safe option other than to create the array using
        // pointer arithmetic and dynamic memory. There are multiple ways to
        // achieve this beyond what I have shown here.
        char **Read_Buffer;  // Array to hold the content of the text file read.
        if((Read_Buffer = malloc((Total_Lines +1) * sizeof(char *))) != NULL)
            {
            for(cnt1=0; cnt1 < Total_Lines; cnt1++)
                {
                if((Read_Buffer[cnt1] = malloc((Max_Width +4) * sizeof(char))) !=
NULL)
                    {
                    for(cnt2=0; cnt2 < Max_Width +4; cnt2++)
                        {
                        // Initialise the array to nul
                        // Note: nul is a character 0 '\0', while null is
                        // a pointer to a non existent object.
                        Read_Buffer[cnt1][cnt2] = '\0';
                        }
                    }
                }
            }
        else
            {
            // This constitutes an application failure from which we must close
            // the application. The user should never receive this error! :)
            printf("Error - unable to allocate required memory for array.\n");
            S_Pause();
            return -1;
            }



        // Next we need to read each line into our Read_Buffer and remove
        // the New line chars '\n'.
        int cnt_lines = 0;  // track array line position.
        while(fgets(string_Temp_Buffer, Max_Width +4, fpFileIn) != NULL)
            {
            // Walk each line from the file (returns the string
            // with '\n' at the end).
            // Strip the newline character from the line and
```

```
            // replace newline char '\n' '\r' with '\0'.
            //string_Temp_Buffer[strcspn(string_Temp_Buffer, "\r\n")] = '\0';
            // Copy the cleaned line of text into our array.
            strcpy(Read_Buffer[cnt_lines], string_Temp_Buffer);
            cnt_lines++;  // move to next line and repeat.
            }

        // It is important to free up resources as soon as they are no longer
required.
        fclose(fpFileIn);  // finished file reads, close the file.

        // Walk through the List and printf each line as text.
        //int cnt1;
        // Note! C Arrays are just integer pointers to the data in memory so we
        // can't accurately test the length of the array at runtime.
        for(cnt1=0; cnt1 < Total_Lines; cnt1++)
            {
            printf("%s", Read_Buffer[cnt1]);
            }
        printf("\n");  // End of line, next line.

        // Important to always "free" the memory as soon as we are finished with
it.
        // Not doing so will lead to a memory leak as a new block of memory will
be
        // created on the heap each time the array is used.
        free(Read_Buffer);
        free(string_Temp_Buffer);
        }  // END file open if, else test.
    S_Pause();
    Con_Clear();
    return 0;
    }

int Mandelbrot_Fractals_Console_ASCII(void)
    {
    /*
    Credits:
    https://cs.nyu.edu/~perlin/
    Ken Perlin
    Professor of Computer Science
    NYU Future Reality Lab F
    >
    Original Source:
    main(k){float i,j,r,x,y=-16;while(puts(""),y++<15)for(x
    =0;x++<84;putchar(" .:-;!/>)|&IH%*#"[k&15]))for(i=k=r=0;
    j=r*r-i*i-2+x/25,i=2*r*i+y/10,j*j+i*i<11&&k++<111;r=j);}
    >
    Info on Mandelbrot sets
    https://mathworld.wolfram.com/MandelbrotSet.html
    */

    // Although this describes a series of planes in 3D layers, the calculations
    // are graphed to a 2D plane and use colour depth to describe points in the
    // lower layers (planes).

    Con_Clear();
    int k = 1;  // First print character; default = 1 (0 to leave blank).
    char colours[] = " .:-;!/>)|&IH%*#";  // 16 colours
```

```
    float i = 0;
    float j = 0;
    float r = 0;
    float x = 0;
    float y = -16;

    // zoom_x, zoom_y are relative and both must be changed as a percentage.
    float zoom_x = 25.00;  // Default = 25,+zoom-In/-zoom-Out
    float zoom_y = 10.00;  // Default = 10,+zoom-In/-zoom-Out

    float offset_x = -2.00;  // Default = -2.00, -pan-L/+pan-R
    float offset_y = 0.00;  // Default = 0.00, -pan-U/+pan-D

    while(y < 15)  // Loop #1
        {
        y++;
        puts("");  // Line break '\n'.

        for(x = 0; x < 88; x++)  // Loop #2, (<84 == the screen print width.)
            {
            // Select colour level (Bitwise AND) from 16 colours, then print.
            putchar(colours[k&15]);

            i=k=r=0;
            while(1)  // Loop #3
                {
                // Calculate x fractal.
                j = ((r*r) - (i*i) + ((x/zoom_x) + offset_x));
                // Calculate y fractal.
                i = ((2*r*i) + ((y/zoom_y) + offset_y));

                // Test for x,y divergence to infinity (lemniscates).
                // In a sense this relates to the period between depth layers
                // and the scale at which they diverge to infinity.
                // The default values offer the most visually appealing balance,
                // meaning they are easier for our brain to interpret.
                if(j*j+i*i > 11)  // Default = 11
                    {
                    break;
                    }

                // Test depth level (Colour).
                k++;
                if(k > 111)  // Default = 111.
                    {
                    break;
                    }

                r=j;  // Start next calculation from current fractal.
                }
            }
        }
    S_Pause();
    Con_Clear();
    return 0;
    }
```

```
// --> START helper functions

// System alert sound (Bell '\a')
int Sys_Sound(void)
    {
    if(OS_Windows == 1)
        {
        //printf("%c", '\a');
        system("rundll32 user32.dll,MessageBeep");
        //system("rundll32.exe Kernel32.dll,Beep 750,300");
        }
    else if(OS_Unix == 1)
        {
        system("paplay /usr/share/sounds/ubuntu/notifications/Blip.ogg");
        //system("paplay /usr/share/sounds/ubuntu/notifications/Rhodes.ogg");
        //system("paplay /usr/share/sounds/ubuntu/notifications/Slick.ogg");
        //system("paplay /usr/share/sounds/ubuntu/notifications/'Soft
delay.ogg'");
        //system("paplay /usr/share/sounds/ubuntu/notifications/Xylo.ogg");
        }
    else
        {
        printf("%c", '\a');
        }
    return 0;
    }

// A simple message box for debugging.
int DebugMsg(char *aTitle, char *aMessage)
    {

    if(OS_Windows)
        {
        // Requires:winuser.h (include Windows.h),User32.dll
        // Not attached to parent console window.
        // https://docs.microsoft.com/en-us/windows/win32/api/winuser
        // /nf-winuser-messageboxa
        #ifdef __WIN32__
        int reterr = 0;
        // May throw a compiler warning... MessageBoxA Not found.
        reterr = MessageBoxA(0, aMessage, aTitle, 0);  // 65536, MB_SETFOREGROUND
        if(reterr == 0)
            {
            // MessageBox() Fail
            return -1;
            }
        #endif // __WIN32__
        // Attached to parent console window.
        //MessageBoxA(FindWindowA("ConsoleWindowClass", NULL),msg,title,0);
        }
    else if(OS_Unix)
        {
        // http://manpages.ubuntu.com/manpages/trusty/man1/xmessage.1.html
        // apt-get install x11-utils
        //system("xmessage -center 'Hello, World!'");
        // Else try wayland
        // https://github.com/Tarnyko/wlmessage
        //system("wlmessage 'Hello, World!'");
        int reterr = 0;
```

```
        char Buffer[128] = {'\0'};
        char Buf_Msg[128] = {'\0'};
        // Place title text in 'apostrophe'.
        strcpy(Buf_Msg, "\'");
        strcat(Buf_Msg, aTitle);
        strcat(Buf_Msg, "\'");

        // Build our command line statement.
        // xmessage [-options] [message ...]
        strcpy(Buffer, "xmessage -center -title ");
        strcat(Buffer, Buf_Msg);
        strcat(Buffer, " \'.:|" );
        strcat(Buffer, aMessage );
        // NOTE! ">>/dev/null 2>>/dev/null" suppresses the console output.
        strcat(Buffer, "|:.\' >>/dev/null 2>>/dev/null" );
        // Send it to the command line.
        reterr = system(Buffer);
        if(reterr != 0) && (reterr != 1) // xmessage failed or not exist.
            {
            // Try Wayland compositor wlmessage.
            strcpy(Buffer, "wlmessage \' |" );
            strcat(Buffer, aMessage );
            strcat(Buffer, "| \' >>/dev/null 2>>/dev/null" );
            reterr = system(Buffer);
            if(reterr != 0) && (reterr != 1)
                {
                // Popup message failed.
                //printf("%d\n", reterr);
                return -1;
                }
            }
        return 0;
        }

    return 0;
    }

// Console Clear
int Con_Clear(void)
    {
    // The system() call allows the programmer to run OS CLI batch commands.
    // It is discouraged as there are more appropriate C functions for most tasks.
    // I am only using it in this instance to avoid invoking additional OS API
    // headers and code.
    if(OS_Windows)
        {
        system("cls");
        }
    else if(OS_Unix)
        {
        system("clear");
        }
    return 0;
    }

// Safe Pause
void S_Pause(void)
    {
    // This function is referred to as a wrapper for S_getchar()
```

```
    printf("\nPress [Enter] to continue...");
    S_getchar();  // Uses S_getchar() for safety.
    }

// Safe getcar() removes all artefacts from the stdin buffer.
int S_getchar(void)
    {
    // This function is referred to as a wrapper for getchar()
    int i = 0;
    int ret;
    int ch;
    // The following enumerates all characters in the buffer.
    while((ch = getchar()) != '\n' && ch != EOF )
        {
        // But only keeps and returns the first char.
        if (i < 1)
            {
            ret = ch;
            }
        i++;
        }
    return ret;
    }

// Console Input (stdin).
// buf is the return buffer, str is printed to the screen.
char *Input(char *buf, char *str, int n)
    {
    FILE *stream = stdin;
    char *empty = "";
    int ret;
    ret = strcmp(str, empty);
    if(ret != 0)  // Don't print an empty string.
        {
        printf("%s", str);  // Input Message
        }
    return S_fgets(buf, n, stream);
    }

// Safe fgets() removes all artefacts from the stdin buffer.
// buf must be at least n + 1 for '\0'
char *S_fgets(char *buf, int n, FILE *stream)
    {
    int i = 0;
    int ch;
    //memset(buf, 0, n);
    // The following enumerates all characters in the buffer.
    while((ch = getc(stream)) != '\n' && ch != EOF )
        {
        // But only keeps and returns n chars.
        if (i < n)
            {
            buf[i] = ch;
            }
        i++;
        }
    buf[i] = '\0';
    return buf;
    }
```

```
// Wrapper for the 2 functions time/date.
// We can create convenience wrapper functions to "wrap" a set of more
// complex tasks in a single function call.
// This is helpful if it is a common set of tasks that are called regularly
// throughout an application.
/*
#include<stdio.h>
#include<time.h>
void datetime(void)
    {
        time_t t;
        time(&t);
        printf(ctime(&t));
    }
*/
int Show_Time_Date(void)
    {
/*
struct tm {
int tm_sec; // seconds, range 0 to 59
int tm_min; // minutes, range 0 to 59
int tm_hour; // hours, range 0 to 23
int tm_mday; // day of the month, range 1 to 31
int tm_mon; // month, range 0 to 11
int tm_year; // The number of years since 1900
int tm_wday; // day of the week, range 0 to 6
int tm_yday; // day in the year, range 0 to 365
int tm_isdst; // daylight saving time
};
*/
    show_time();
    printf(" - ");
    show_date();
    printf("\n");
    return 0;
    }

// Display current system time.
int show_time(void)
    {
    // Windows Command-line
    //system("ECHO %time% - %date%");
    //system("TIME /T");
    // Unix command-line
    //system("date +%T");

    time_t t = time(NULL);
    struct tm tm = *localtime(&t);
    printf("%02d:%02d:%02d", tm.tm_hour, tm.tm_min, tm.tm_sec);

    return 0;
    }

// Display current system date.
int show_date(void)
    {
    // Windows command-line
    //system("ECHO %time% - %date%");
```

```
    //system("DATE /T");
    // Unix command-line
    //system("date +%F");

    time_t t = time(NULL);
    struct tm tm = *localtime(&t);
    printf("%d-%02d-%02d", tm.tm_year + 1900, tm.tm_mon + 1, tm.tm_mday);

    return 0;
    }
```

Language: FreeBASIC

Code: “Functions.bas”

```
'------------------------------------------------------------------------------
' Name:         Functions.bas
' Purpose:      Examples
'
' Platform:     Win64, Ubuntu64
'
' Author:       Axle
' Created:      22/02/2022
' Updated:      19/05/2022
' Copyright:    (c) Axle 2022
' Licence:      MIT No Attribution
'------------------------------------------------------------------------------
' Test if Windows or Unix OS
#ifdef __FB_WIN32__
#include once "windows.bi"
#define OS_Windows 1   ' 1 = True (aka Bool)
#define OS_Unix 0
#endif

#ifdef __FB_UNIX__'__FB_LINUX__
' TODO
#define OS_Unix 1
#define OS_Windows 0   ' 0 = False (aka Bool)
#endif

' Define extra functions so we can place them at the bottom of the page.
Declare Function main_procedure() As Integer
Declare Function Menu_Routine() As Integer
Declare Function Stick_Animation() As Integer
Declare Function File_Write_Example() As Integer
Declare Function File_Read_Example1() As Integer
Declare Function File_Read_Example2() As Integer
Declare Function Mandelbrot_Fractals_Console_ASCII() As Integer


Declare Function Sys_Sound() As Integer
Declare Function DebugMsg(Byref title As String, Byref msg As String) As Integer
Declare Sub Clear_Stdin()
Declare Function Con_Clear() As Integer
Declare Function Con_Pause() As Integer   ' GetKey Version
Declare Function Show_Time_Date() As Integer
Declare Function show_time() As Integer
Declare Function show_date() As Integer
```

```
main_procedure()

Function main_procedure() As Integer  ' Main procedure

    Menu_Routine()

    Con_Pause() ' DEBUG Pause
    Return 0
End Function  ' END main_procedure <---

' --> START Application Functions

Function Menu_Routine() As Integer
    While(1)
        ' Forever loop. Needs a break, return, or exit() statement to
        ' exit the loop.FreeBASIC = Exit While
        ' Placing the menu inside of the while loop "Refreshes" the menu
        ' if an incorrect value is supplied.
        Con_Clear()
        ' Clearing the keyboard buffer is me being pedantic :)
        ' FB uses the GC Compiler and MinGW(32) which can leave
        ' stray '\r''\n' in the stdin buffer.
        Clear_Stdin() ' Clear the keyboard buffer
        Dim As Integer options = 0  ' Menu variable.

        Print "============================================="
        Print "  MAIN MENU"
        Print "============================================="
        Print "  1 - Stick Animation"
        Print "  2 - File Write Example"
        Print "  3 - File Read Example 1"
        Print "  4 - File Read Example 2"
        Print "  5 - Debug Message Box"
        Print "  6 - System Sound Test"
        Print "  7 - Mandelbrot Fractals"
        Print "  8 - Display time and date"
        Print ""
        Print "  9 - Exit The Application"
        Print "---------------------------------------------"
        Clear_Stdin()
        Input ; "  Enter your menu choice: ", options
        Print ""

        ' Check what choice was entered and act accordingly
        ' We can add as many choices as needed
        If options = 0 Then
            ' Ignore false [Enter]s
        Elseif options = 1 Then
            Stick_Animation()
        Elseif options = 2 Then
            File_Write_Example()
        Elseif options = 3 Then
            File_Read_Example1()
        Elseif options = 4 Then
            File_Read_Example2()
        Elseif options = 5 Then
            ' Send String. "Hello"
            Dim As String aVariable = "This is my message."
```

```
            DebugMsg("DEBUGmsg", aVariable)
            ' Send  Int 125
            'dim as Integer iVariable = 125
            'DebugMsg("DEBUGmsg", Str(iVariable))
            ' See also Val(), ValInt()
        Elseif options = 6 Then
            Sys_Sound()
        Elseif options = 7 Then
            Mandelbrot_Fractals_Console_ASCII()
        Elseif options = 8 Then
            Show_Time_Date()
            Con_Pause()
            Con_Clear()
        Elseif options = 9 Then
            Print "Exiting the application..."
            Sleep 1000  ' Allow 1 second for the Exit notice to display.
            Exit While
        Else
            Sys_Sound()
            Print !"Invalid option.\nPlease enter a number from the Menu
Options.\n"
            Sleep 1000
        End If
    Wend

    Return 0
End Function

Function Stick_Animation() As Integer
    ' Stick man animation
    Dim As Integer Len_Animation = 11  ' Keep a record of the array length.
    Dim As String Animation(11) => { _
    !"   o           \n" _
    !"  /|\\          \n" _
    !"  / \\          ", _
    !" \\ o /         \n" _
    !"   |           \n" _
    !"  / \\          ", _
    !"  _ o          \n" _
    !"   /\\          \n" _
    !"  | \\          ", _
    !"               \n" _
    !"   ___\\o       \n" _
    !"  |)  |        ", _
    !"    __|        \n" _
    !"      \\o       \n" _
    !"      ( \\      ", _
    !"      \\ /      \n" _
    !"       |       \n" _
    !"      /o\\      ", _
    !"         |__   \n" _
    !"       o/      \n" _
    !"      / )      ", _
    !"               \n" _
    !"        o/__   \n" _
    !"        |  (\\  ", _
    !"           o _ \n" _
    !"           /\\  \n" _
    !"           / | ", _
```

```
    !"          \\ o /\n" _
    !"            |  \n" _
    !"           / \\ ", _
    !"            o  \n" _
    !"           /|\\ \n" _
    !"           / \\ "}

    Dim As Integer a = 0
    Dim As Integer b = 0
    Dim As Integer repeat = 3
    Dim As Integer Speed = 400

    While(a < repeat)
        For b = 0 To Len_Animation -1 Step 1  ' Loop animation forward
            Con_Clear()
            Print Animation(b)
            Sleep(Speed)
        Next b
        For b = Len_Animation -1 To 0 Step -1  ' Loop animation backwards
            Con_Clear()
            Print Animation(b)
            Sleep(Speed)
        Next b
        a+= 1
        Sleep(Speed)
    Wend
    Con_Clear()
    Return 0
End Function

Function File_Write_Example() As Integer

    Const filename As String = "MyTextFile.txt" ' Output file.
    ' Output file to write to.
    '==> Open Output file for text append ops.
    Dim FileOut As Integer = Freefile()
    If 0 <> Open(filename, For Append, As FileOut) Then
        Print "ERROR! Cannot open Output file " & filename
        Con_Pause() ' Wait until a key is pressed
        Return -1
    End If

    Dim As String string_Temp_Buffer

    Print "Please enter the text you would like to write to file " & filename
    Print "Type a line of text ... followed by [Enter]"
    Input "", string_Temp_Buffer

    ' Write the buffer to a new line of the open file (append).
    Print #FileOut, string_Temp_Buffer
    ' We must always remember to close the file when finished.
    Close #FileOut

    Print !"\nFile write completed..."
    Print "Press [Enter] to return to the MAIN MENU..."
    Con_Pause() 'wait until a key is pressed
    Con_Clear()
    Return 0
End Function
```

```
Function File_Read_Example1() As Integer
    Const filename As String = "MyTextFile.txt"  ' Input file.
    Dim string_Temp_Buffer As String  ' Temporary buffer
    'Dim As Integer Total_Lines = 0 ' = lines in the text file (To be calculated).

    Dim FileIn As Integer = Freefile()
    ' It is possible that the file may not yet exist. Opening it
    ' as "read/Input" will return an error. Let's test if the file exists first.
    If Open(filename, For Input, As #FileIn) <> 0 Then
        Print "ERROR! Cannot open Output file " & filename
        Print "Maybe the file has not yet been created."
        Print "Please select from the MAIN Menu"
        Print "to create a new file."
        Print "Press [Enter] to return to the MAIN MENU..."
        Con_Pause() 'wait until a key is pressed
        Return 0
    Else  ' Continue to process file...
        ' Line Input # will read up to one line (including '\r''\n') at a time.
        While Not Eof (FileIn)
            Line Input #FileIn, string_Temp_Buffer
            Print string_Temp_Buffer
        Wend

        ' It is important to free up resources as soon as they are no longer
required.
        Close #FileIn' finished file reads, close the file.
    End If

    Print !"\nPress [Enter] to return to the MAIN MENU..."
    Con_Pause()  'wait until a key is pressed
    Con_Clear()
    Return 0
End Function

Function File_Read_Example2() As Integer
    Const filename As String = "MyTextFile.txt"  ' Input file.

    ' in FreeBASIC we also have to allocate enough char space to hold the string
    ' values to be stored. Because we don't know in advance how large the file
    ' will be, we also have to create a dynamic array in memory "On the heap"
    ' at run time.

    Dim As Integer cnt1  ' To enumerate and set array parameters.
    Dim As Integer Total_Lines = 0  ' To build our dynamic array size.

    Dim FileIn As Integer = Freefile()
    ' It is possible that the file may not yet exist. Opening it
    ' as "read/Input" will return an error. Let's test if the file exists first.
    If Open(filename, For Input, As #FileIn) <> 0 Then
        Print "ERROR! Cannot open Output file " & filename
        Print "Maybe the file has not yet been created."
        Print "Please select from the MAIN Menu"
        Print "to create a new file."
        Print "Press [Enter] to return to the MAIN MENU..."
        Con_Pause()  'wait until a key is pressed
        Return 0

    Else  ' Continue to process file...
```

```
        ' Create a temp buffer to hold each line.
        ' In free basic Type String is dynamically resized up to 2GiB
        Dim As String String_Temp_Buffer

        ' Check the file for the number of lines.
        ' Line Input # will read one line (up to '\r''\n') at a time.
        While Not Eof (FileIn)
            Line Input #FileIn, String_Temp_Buffer
            Total_Lines = Total_Lines + 1
        Wend
        Seek #FileIn, 1  ' Set pointer back to the start of the file.

        ' Build our dynamic array here.
        ' Now that we now how many lines to allocate, we can create a suitable
        ' sized array to hold the contents. Using ReDim we can set the required
        ' size of the dynamic array on the heap at runtime.
        ' Array declared in dynamic memory (On the heap).
        Dim As String Read_Buffer(Any)
        Redim Read_Buffer(Total_Lines)

        ' Read the file into the Dynamic Array at its correct line location,
        ' removing the new line chars '\r''\n'.
        For cnt1 = 0 To Total_Lines Step 1
            ' add each value to the correct array position by (Line/cnt1).
            Line Input #FileIn, Read_Buffer(cnt1)
        Next cnt1

        ' It is important to free up resources as soon as they are no longer
required.
        Close #FileIn  ' finished file reads, close the file.

        ' Print the lines from the array.
        For cnt1 = 0 To Total_Lines Step 1
            Print Read_Buffer(cnt1)
        Next cnt1

        Print !"\nPress [Enter] to return to the MAIN MENU..."
        Con_Pause()  'wait until a key is pressed

        ' Important to always "free" the memory as soon as we are finished with
it.
        ' Not doing so will lead to a memory leak as a new block of memory will
        ' be created on the heap each time the dynamic array is used.
        ' I am uncertain if dynamic memory (On the Heap) is freed when the
        ' Function exits, so I am just playing safe ¯\_(''/)_/¯
        Erase Read_Buffer
    End If' END file open if, else test.
    Con_Clear()
    Return 0
End Function


Function Mandelbrot_Fractals_Console_ASCII() As Integer
    /'
    Credits:
    https://cs.nyu.edu/~perlin/
    Ken Perlin
    Professor of Computer Science
    NYU Future Reality Lab F
```

```
    >
    Original Source:
    main(k){float i,j,r,x,y=-16;While(puts(""),y++<15)For(x
    =0;x++<84;putchar(" .:-;!/>)|&IH%*#"[k&15]))For(i=k=r=0;
    j=r*r-i*i-2+x/25,i=2*r*i+y/10,j*j+i*i<11&&k++<111;r=j);}
    >
    Info On Mandelbrot sets
    https://mathworld.wolfram.com/MandelbrotSet.html
    '/

    ' Although this describes a series of planes in 3D layers, the calculations
    ' are graphed to a 2D plane and use colour depth to describe points in the
    ' lower layers (planes).

    Con_Clear()
    Dim As Integer k = 1  ' First print character; default = 1 (0 to leave blank).
    Dim As String colours = " .:-;!/>)|&IH%*#"  ' 16 colours

    Dim As Single i=0
    Dim As Single j=0
    Dim As Single r=0
    Dim As Single x = 0
    Dim As Single y = -16

    ' zoom_x, zoom_y are relative and both must be changed as a percentage.
    Dim As Single zoom_x = 25.00  ' Default = 25,+zoom-In/-zoom-Out
    Dim As Single zoom_y = 10.00  ' Default = 10,+zoom-In/-zoom-Out

    Dim As Single offset_x = -2.00  ' Default = -2.00, -pan-L/+pan-R
    Dim As Single offset_y = 0.00  ' Default = 0.00, -pan-U/+pan-D

    While(y < 15)  ' Loop #1
        y+= 1
        Print ""  ' Line break '\n'.

        For x = 0 To 88-1 Step 1  ' Loop #2, (<84 == the screen print width.)
            ' Select colour level (Bitwise AND) from 16 colours, then print.
            Print Chr(colours[k And 15]);

            i=0
            k=0
            r=0
            While(1)  ' Loop #3
                ' Calculate x fractal.
                j = ((r*r) - (i*i) + ((x/zoom_x) + offset_x))
                ' Calculate y fractal.
                i = ((2*r*i) + ((y/zoom_y) + offset_y))

                ' Test for x,y divergence to infinity (lemniscates).
                ' In a sense this relates to the period between depth layers
                ' and the scale at which they diverge to infinity.
                ' The default values offer the most visually appealing balance,
                ' meaning they are easier for our brain to interpret.
                If(j*j+i*i > 11) Then  ' Default = 11
                    Exit While
                End If

                ' Test depth level (Colour).
                k+= 1
```

```
                If(k > 111) Then  ' Default = 111.
                    Exit While
                End If

                r=j  ' Start next calculation from current fractal.
            Wend
        Next x
    Wend
    Con_Pause()
    Con_Clear()
    Return 0
End Function


' --> START helper functions

Function Sys_Sound() As Integer

    If(OS_Windows = 1) Then
        'Print !"\a"
        Shell "rundll32 user32.dll,MessageBeep"
        'Shell "rundll32.exe Kernel32.dll,Beep 750,300"
    Elseif(OS_Unix = 1) Then
        Shell "paplay /usr/share/sounds/ubuntu/notifications/Blip.ogg"
        'Shell "paplay /usr/share/sounds/ubuntu/notifications/Rhodes.ogg"
        'Shell "paplay /usr/share/sounds/ubuntu/notifications/Slick.ogg"
        'Shell "paplay /usr/share/sounds/ubuntu/notifications/'Soft delay.ogg'"
        'Shell "paplay /usr/share/sounds/ubuntu/notifications/Xylo.ogg"
    Else
        Print !"\a"
    End If
    Return 0
End Function

Function DebugMsg(Byref aTitle As String, Byref aMessage As String) As Integer
    ' Requires:"windows.bi" (#include once "windows.bi").
    ' Not attached to parent console window.
    ' https://docs.microsoft.com/en-us/windows/win32/api/winuser
    ' /nf-winuser-messageboxa
    ' May throw a compiler warning... MessageBoxA Not found.
    If(OS_Windows = 1) Then
        #ifdef __FB_WIN32__
            MessageBox(NULL, aMessage, aTitle, MB_OK)  ' MB_OK|MB_SETFOREGROUND
        #endif
    Elseif(OS_Unix = 1) Then
        ' http://manpages.ubuntu.com/manpages/trusty/man1/xmessage.1.html
        ' apt-get install x11-utils
        'system("xmessage -center 'Hello, World!'");
        ' Else try wayland
        ' https://github.com/Tarnyko/wlmessage
        'system("wlmessage 'Hello, World!'");
        Dim As Integer reterr = 0
        Dim As String Buffer = ""
        Dim As String Buf_Msg = ""
        ' Place title text in 'apostrophe'.
        Buf_Msg = "\'" & aTitle & "\'"

        ' Build our command line statement.
        ' xmessage [-options] [message ...]
```

```
        ' NOTE! ">>/dev/null 2>>/dev/null" suppresses the console output.
        Buffer = "xmessage -center -title " & Buf_Msg & " '.:|" & aMessage & "|:.'
>>/dev/null 2>>/dev/null"

        ' Send it to the command line.
        reterr = Shell(Buffer)
        If(reterr <> 0) And (reterr <> 1) Then  ' xmessage failed or not exist.
            ' Try Wayland compositor wlmessage.
            Buffer = "wlmessage ' |" & Str(aMessage) & "| ' >>/dev/null
2>>/dev/null"
            reterr = Shell(Buffer)
            If(reterr <> 0) And (reterr <> 1) Then
                ' Popup message failed.
                'printf("%d\n", reterr);
                Return -1
            End If
        End If

        Return 0
    Else
        ' OS Unknown
    End If
    Return 0
End Function

' A wrapper to flush/clear the keyboard input buffer
Sub Clear_Stdin()
    While Inkey <> ""  ' loop until the Inkey buffer is empty
    Wend
End Sub

' Console Clear
Function Con_Clear() As Integer
    ' The system() call allows the programmer to run OS command line batch
commands.
    ' It is discouraged as there are more appropriate C functions for most tasks.
    ' I am only using it in this instance to avoid invoking additional OS API
headers and code.
    If (OS_Windows) Then
        Shell "cls"
    Elseif (OS_Unix) Then
        Shell "clear"
    End If
    Return 0
End Function

' Console Pause (GetKey version)
Function Con_Pause() As Integer
    Dim As Long dummy
    Print !"\nPress any key to continue..."
    dummy = Getkey
    Return 0
End Function

' Wrapper for the 2 functions time/date.
' We can create convenience wrapper functions to "wrap" a set of more
' complex tasks in a single function call.
' This is helpful if it is a common set of tasks that are called regularly
' throughout an application.
```

```
Function Show_Time_Date() As Integer
    show_time()
    Print " - ";
    show_date()
    Print ""
    Return 0
End Function

' Display current system time.
Function show_time() As Integer
    Print Time;
    Return 0
End Function

' Display current system date.
Function show_date() As Integer
    ' Note: FB returns format mm-dd-yyyy
    ' This is due to the QBasic American origins.
    ' You can use the C functions in the C example to change this.
    Print Date;
    Return 0
End Function
```

Language: Python 3

Code: “Functions.py”

```
#-------------------------------------------------------------------------------
# Name:        Functions.py
# Purpose:     Examples
#
# Platform:    REPL*, Win64, Ubuntu64
#
# Author:      Axle
# Created:     22/02/2022
# Updated:
# Copyright:   (c) Axle 2022
# Licence:     MIT No Attribution
#-------------------------------------------------------------------------------
# * This example is best run in a native OS console window.
#-------------------------------------------------------------------------------


def main():

    Menu_Routine()


    Con_Pause()  # DEBUG Pause
    return None
    # END Main() <---

# --> START Application Functions

def Menu_Routine():
    while(True):
        # Forever loop. Needs a break, return, or exit() statement to
        # exit the loop.
```

```
        Con_Clear()
        # Placing the menu inside of the while loop "Refreshes" the menu
        # if an incorrect value is supplied.

        option = ""  # Menu variable.

        print("================================================")
        print("  MAIN MENU")
        print("================================================")
        print("  1 - Stick Animations")
        print("  2 - File Write Example")
        print("  3 - File Read Example 1")
        print("  4 - File Read Example 2")
        print("  5 - Debug Msg Box")
        print("  6 - System Sound Test")
        print("  7 - Mandelbrot Fractals")
        print("  8 - Show time and date")
        print("")
        print("  9 - Exit The Application")
        print("------------------------------------------------")
        options = input("  Enter your menu choice: ")
        print("")

        # Check what choice was entered and act accordingly.
        # We can add as many choices as needed.
        if options == '0':
            pass# Ignore false [Enter]s
        elif options == '1':
            Stick_Animations()
        elif options == '2':
            File_Write_Example()
        elif options == '3':
            File_Read_Example1()
        elif options == '4':
            File_Read_Example2()
        elif options == '5':
            # Send String. "Hello"
            aVariable = "This is my message."
            DebugMsg("DEBUGmsg", aVariable);
            # Send  Int 125
            #iVariable = 125
            #DebugMsg("DEBUGmsg", str(iVariable))
        elif options == '6':
            Sys_Sound()
        elif options == '7':
            Mandelbrot_Fractals_Console_ASCII()
        elif options == '8':
            Show_Time_Date()
            Con_Pause()  # wait until a key is pressed
            Con_Clear()
        elif options == '9':
            print("Exiting the application...")
            Con_sleep(1)  # Allow 1 second for the Exit notice to display.
            break
        else:
            Sys_Sound()
            print("Invalid option.\nPlease enter a number from the Menu
Options.\n")
            Con_sleep(1)
```

```
    return None

# --> START Application Functions
def Stick_Animations():
    # Stick man animation
    Len_Animation = 11  # Keep a record of the list length.
    Animation = [
    "   o           \n"
    "  /|\\          \n"
    "  / \\          ",
    " \\ o /         \n"
    "   |           \n"
    "  / \\          ",
    "  _ o          \n"
    "   /\\          \n"
    "  | \\          ",
    "               \n"
    "   ___\\o       \n"
    "  |)  |        ",
    "    __|        \n"
    "      \\o       \n"
    "      ( \\      ",
    "      \\ /      \n"
    "       |       \n"
    "      /o\\      ",
    "         |__   \n"
    "       o/      \n"
    "      / )      ",
    "               \n"
    "        o/__   \n"
    "        |  (\\  ",
    "           o _ \n"
    "           /\\  \n"
    "           / | ",
    "          \\ o /\n"
    "            |  \n"
    "           / \\ ",
    "            o  \n"
    "           /|\\ \n"
    "           / \\ "]

    a = 0
    b = 0
    repeat = 3
    Speed = 0.3

    while(a < repeat):
        for b in range(0, Len_Animation):  # Loop animation forward.
            Con_Clear()
            print(Animation[b])
            Con_sleep(Speed)
        Con_sleep(Speed)
        for b in range(Len_Animation-1, -1, -1):  # Loop animation backwards.
            Con_Clear()
            print(Animation[b])
            Con_sleep(Speed)
        a+= 1
        Con_sleep(Speed)
```

```
    Con_Clear()
    return None

def File_Write_Example():

    filename = "MyTextFile.txt"
    # Output file to write to.
    # ==> Open Output file for text append ops
    string_Temp_Buffer = ""

    with open(filename, "a") as outfile:
        print("Please enter the text you would like to write to file " + filename)
        print("Type a line of text ... followed by [Enter]")
        string_Temp_Buffer = input()

        # append a new line character \n
        string_Temp_Buffer = string_Temp_Buffer + '\n'
        # Write/append 'a' string_Temp_Buffer to file.
        outfile.write(string_Temp_Buffer)

    print("\nFile write completed...")
    Con_Pause()  # wait until a key is pressed
    Con_Clear()
    return None

# A simple routine that reads a files contents directly to the screen
# using a single buffer.
def File_Read_Example1():
    filename = "MyTextFile.txt"  # Input file.
    string_Temp_Buffer = ""

    # It is possible that the file may not yet exist. Opening it
    # as "r" will return an exception. Let's test if the file exists first.
    try:
        with open(filename, "r") as infile:  # Open the CSV File.
            for string_Temp_Buffer in infile:
                # Strip the newline character from the line.
                string_Temp_Buffer = string_Temp_Buffer.strip('\n')
                print(string_Temp_Buffer)

        print("")
        Con_Pause()  # wait until a key is pressed
        Con_Clear()
        return None

    except FileNotFoundError:  # If the file does not yet exist.
        print("\nERROR! Cannot open Output file " + filename)
        print("Maybe the file has not yet been created.")
        print("Please select from the MAIN Menu")
        print("to create a new file.")
        Con_Pause()
        return None


def File_Read_Example2():

    filename = "MyTextFile.txt"
    # main Data List.
    Read_Buffer = []
```

```
    # It is possible that the file may not yet exist. Opening it
    # as "r" will return an exception. Let's test if the file exists first.
    try:
        with open(filename, "r") as file:  # Open the CSV File.
            # Walk each line from the file.
            for buffer in file:
                # Strip the newline character from the line.
                buffer = buffer.strip('\n')
                # Append each line to the list.
                Read_Buffer.append(buffer)

        # Walk through the List and print each line as text.
        for cnt1 in range(len(Read_Buffer)):
            # ' Print the lines from the array.
            print(Read_Buffer[cnt1])

        print("")
        Con_Pause()  # wait until a key is pressed
        Con_Clear()
        return None

    except FileNotFoundError:  # If the file does not yet exist.
        print("The CSV file has not yet been created.")
        print("Please select Option 1 from the MAIN Menu")
        print("to start the data entry.")
        input("Press [Enter] to return to the MAIN MENU...")
        return None


def Mandelbrot_Fractals_Console_ASCII():

    # Credits:
    # https://cs.nyu.edu/~perlin/
    # Ken Perlin
    # Professor of Computer Science
    # NYU Future Reality Lab F
    # >
    # Original Source:
    # main(k){float i,j,r,x,y=-16;while(puts(""),y++<15)for(x
    # =0;x++<84;putchar(" .:-;!/>)|&IH%*#"[k&15]))for(i=k=r=0;
    # j=r*r-i*i-2+x/25,i=2*r*i+y/10,j*j+i*i<11&&k++<111;r=j);}
    # >
    # Info on Mandelbrot sets
    # https://mathworld.wolfram.com/MandelbrotSet.html

    # Although this describes a series of planes in 3D layers, the calculations
    # are graphed to a 2D plane and use colour depth to describe points in the
    # lower layers (planes).

    Con_Clear()
    k = 1  # First print character; default = 1 (0 to leave blank).
    colours = " .:-;!/>)|&IH%*#"  # 16 colours

    i = 0.0
    j = 0.0
    r = 0.0
    x = 0.0
    y = -16
```

```
    # zoom_x, zoom_y are relative and both must be changed as a percentage.
    zoom_x = 25.00  # Default = 25,+zoom-In/-zoom-Out
    zoom_y = 10.00  # Default = 10,+zoom-In/-zoom-Out

    offset_x = -2.00  # Default = -2.00, -pan-L/+pan-R
    offset_y = 0.00  # Default = 0.00, -pan-U/+pan-D

    while(y < 15):  # Loop #1
        y+= 1
        print("")  # Line break '\n'.

        for x in range(0, 88 -1):  # Loop #2, (<84 == the screen print width.)
            # Select colour level (Bitwise AND) from 16 colours, then print.
            print (colours[(k & 15)], end= "")

            i=0
            k=0
            r=0
            while(1):  # Loop #3
                # Calculate x fractal.
                j = ((r*r) - (i*i) + ((x/zoom_x) + offset_x))
                # Calculate y fractal.
                i = ((2*r*i) + ((y/zoom_y) + offset_y))

                # Test for x,y divergence to infinity (lemniscates).
                # In a sense this relates to the period between depth layers
                # and the scale at which they diverge to infinity.
                # The default values offer the most visually appealing balance,
                # meaning they are easier for our brain to interpret.
                if(j*j+i*i > 11):  # Default = 11
                    break
                # Test depth level (Colour).
                k+= 1
                if(k > 111):  # Default = 111.
                    break

                r=j  # Start next calculation from current fractal.
            # End Loop #3
        # End Loop #2
    # End Loop #1
    Con_Pause()
    Con_Clear()
    return None


# --> START helper functions

# A Popup Message Box
def DebugMsg(aTitle, aMessage ):

    import sys

    # for windows
    if sys.platform.startswith('win32'):
        import win32api
        win32api.MessageBox(0, aMessage, aTitle, 0)  # 65536 = MB_SETFOREGROUND
        # MessageBox[W] is for Unicode text, [A] is for ANSI text.
        ## Alternative
```

```
        #import ctypes
        #ctypes.windll.user32.MessageBoxW(0, aMessage, aTitle, 0)
        return 0
    # for linux
    elif sys.platform.startswith('linux'):
        import os
        # http://manpages.ubuntu.com/manpages/trusty/man1/xmessage.1.html
        # apt-get install x11-utils
        #system("xmessage -center 'Hello, World!'");
        # Else try wayland
        # https://github.com/Tarnyko/wlmessage
        #system("wlmessage 'Hello, World!'");
        reterr = 0
        Buffer = ""
        Buf_Msg = ""
        # Place title text in 'apostrophe'.
        Buf_Msg = "\'" + aTitle + "\'"

        # Build our command line statement.
        # xmessage [-options] [message ...]
        # NOTE! ">>/dev/null 2>>/dev/null" suppresses the console output.
        Buffer = "xmessage -center -title " + Buf_Msg + " \'.:|" + aMessage +
"|:.\' >>/dev/null 2>>/dev/null"

        # Send it to the command line.
        reterr = os.system(Buffer)
        if(reterr != 0) & (reterr != 1):  # xmessage failed or not exist.
            # Try Wayland compositor wlmessage.
            Buffer = "wlmessage \' |" + aMessage + "| \' >>/dev/null 2>>/dev/null"
            reterr = os.system(Buffer)
            if(reterr != 0) & (reterr != 1):
                # Popup message failed.
                #printf("%d\n", reterr);
                return -1

        return 0
    else:
        pass
        return -1  # Other OS
    return 0

# Test if we are inside of the REPL interactive interpreter.
# This function is in alpha and may not work as expected.
def Con_IsREPL():
    import os
    if os.sys.stdin and os.sys.stdin.isatty():
        if os.isatty(os.sys.stdout.fileno()):
            return 0  # OS Command Line
        else:
            return 1  # REPL - Interactive Linux?
    else:
        return 1  # REPL - Interactive Windows?
    return None

# Cross platform console clear.
# This function is in alpha and may not work as expected.
def Con_Clear():
    # The system() call allows the programmer to run OS command line batch
commands.
```

```
    # It is discouraged as there are more appropriate functions for most tasks.
    # I am only using it in this instance to avoid invoking additional OS API
headers and code.
    import os
    if os.sys.stdin and os.sys.stdin.isatty():
        if os.isatty(os.sys.stdout.fileno()):# Clear function doesn't work in Py
REPL
            # for windows
            if os.name == 'nt':
                os.system('cls')
            # for mac and linux
            elif os.name == 'posix':
                os.system('clear')
            else:
                return None  # Other OS
        else:
            return None  # REPL - Interactive Linux?
    else:
        return None  # REPL - Interactive Windows?
    return None

def Sys_Sound():
    # The system() call allows the programmer to run OS command line batch
    # commands. It is discouraged as there are more appropriate functions
    # for most tasks. I am only using it in this instance to avoid invoking
    # additional OS API headers and code.
    import os
    if os.sys.stdin and os.sys.stdin.isatty():
        if os.isatty(os.sys.stdout.fileno()):
            # for windows
            if os.name == 'nt':
                os.system("rundll32 user32.dll,MessageBeep")
            # for mac and linux
            elif os.name == 'posix':
                os.system("paplay
/usr/share/sounds/ubuntu/notifications/Blip.ogg")
                ## Rhodes.ogg,Slick.ogg,'Soft delay.ogg',Xylo.ogg
            else:  # Other OS
                print("\a", end="")
        else:  # REPL - Interactive Linux?
            os.system("paplay /usr/share/sounds/ubuntu/notifications/Blip.ogg")
    else:  # REPL - Interactive Windows?
        os.system("rundll32 user32.dll,MessageBeep")
    return None

# Console Pause wrapper.
def Con_Pause():
    dummy = ""
    print("")
    dummy = input("Press [Enter] key to continue...")
    return None

# Console sleep
def Con_sleep(times: float):
    import time
    time.sleep(times)
    return None

# Wrapper for the 2 functions time/date.
```

```
# We can can create convenience wrapper functions to "wrap" a set of more
# complex tasks in a single function call.
# This is helpful if it is a common set of tasks that are called regularly
# throughout an application.
def Show_Time_Date():
    show_time()
    print(" - ", end="")
    show_date()
    print("")
    return 0

# Display current system time.
def show_time():
    import time
    t = time.localtime()
    current_time = time.strftime("%H:%M:%S", t)
    print(current_time, end="")
    return None

# Display current system date.
def show_date():
    import time
    d = time.localtime()
    current_date = time.strftime("%Y/%m/%d", d)
    print(current_date, end="")
    return None

if __name__ == '__main__':
    main()
```

Variable: By reference and By Value

*Although this relates more directly to **Variables**, I have placed it after **Functions** as this is where it will come into the most common use. Although this relates to an advanced subject area such as pointers and objects, a brief explanation is required as you will begin to make use of Variables at a beginner's level.*

Variables can be copied in 2 different ways; *By Reference* or *By Value*. A variable is just a label name that points to the data that is associated with it.

The variable **MyData** is the index name for the **Data** that is stored somewhere in memory. We can copy the variable name **MyData** to a new variable **MyData_2** keeping the index (pointer) to the exact same data in memory as **MyData.** This is known as **By Reference** as we only copy the reference (Pointer) to the original data. Making changes to **MyData_2** will also make changes to **MyData** as they both point to (Reference) the same data in memory.

When we copy a variable using *By Value*, we copy the data from the original variable into a new memory location such that we now have 2 separate variables with independent and separate data in memory. When we copy **MyData** by value, we create a new variable name **MyData_2** as well as a new index (pointer) to a new memory location, as well as making a duplicate copy of the data. Making changes to **MyData_2** will have no effect on **MyData** as they point to (Reference) separate data allocation in memory.

C generally defaults to *By Value* for integers and By *Reference* for arrays (which includes all Strings). You will see this Referencing and Dereferencing in C with the use of pointers * and & or lack of * and &.

BASIC, Including FreeBASIC, Python and other high level languages most often default to *By Value* meaning a new copy of the data is created when we make a copy of a variable.

Example working applications

A working “Flat File Database” example with menu, data entry and data retrieval.

Language: C

Code example: “Pract_Task.c”

```
//################################################################
// Name:          Practical task
// Purpose:       Employee package delivery tracker
// Requires:      See individual Modules following
// Author:        Axle, Daniel
// Contributors:  Add name(s) for anyone who might have helped here
// Copyright:     Add copyright info here, if any, such as
//                (c) Axle 2021, Daniel 2021
// Licence:       Add license here you'd like to use, such as creative
//                commons, GPL, Mozilla Public License, etc.
//                e.g. MIT
//                https://creativecommons.org/licenses/by/2.0/au/
// Created:       07/09/2021 < add the date you start your
//                python file
// Last Modified: 20/10/2021 < add the last date you updated your
//                C source file
// Versioning    ("MAJOR.MINOR.PATCH") Such as Version 1.0.0
//################################################################
// NOTES:
// https://www.w3schools.in/c-tutorial/
// https://www.programiz.com/c-programming
// ^ an excellent resource to check as you code
//################################################################
//----------------------------------------------------------------
// NOTES:
// Updated 06/01/2022
// Example csv database.
//
// TODO: strncpy, strncat for string safety
// TODO: Error handling
// TODO: split statements exceeding 80 character width.
//
//----------------------------------------------------------------

//  ---> Include standard libraries
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <synchapi.h> // For Sleep() Recommend windows.h instead.

// ---> Precompiler MACROS
// MACROS are replaced by the statement or routine at compile time and make
// our source easier to read. STRINGSIZE is a little easier to understand than
// a magic number like 64
// The unused() macro simply avoids unused parameter warnings.
#define unused(x) (x) = (x)
// Set MAX string length. scanf() also limits the input by using "%60s" .
```

```c
// Limits input to 60 chars.
#define INPUTSIZE 64
// set MAX string length for building the formatted lines for the csv file.
// employee_packages_delivered[] has 8 elements at MAX 64 characters.
// 8 * 64 = 512 (the minimum safe length to hold all strings from the array).
#define STRINGSIZE 512

// Declare functions used at the bottom of the page (after the formal entry point).
// The interpreter or compiler must read and know all Functions and Subroutines
// that exist before entering the application main routine. Library function, as
// well as our program functions must be read first at the top of the page
// before entering the application. We can let the Interpreter/compiler know
// they exist by declaring the Function at the top of the page, and moving the
// actual function routine to the bottom of the page.
int Enter_Daily_Packages_Delivered();
int Produce_Packages_Delivered_Report();

char *Input(char *str, char *buf, int n, FILE *stream);

// ---> START main application
int main(int argc, char *argv[])
{
    unused(argc);// turns off the compiler warning for unused argc, argv
    unused(argv);// turns off the compiler warning for unused argc, argv
Sleep(4000);
    while(1)
        {
            // Forever loop. Needs a break, return, or exit() statement to
            // exit the loop.
            // Placing the menu inside of the while loop "Refreshes" the menu
            // if an incorrect value is supplied.
            char temp[4] = {'\0'};// Inputs are as char/string.
            int option;  // Declares a menu variable as empty.

            printf("=============================================\n");
            printf("MAIN MENU\n");
            printf("=============================================\n");
            printf("1 - Enter Daily Packages Delivered\n");
            printf("2 - Produce Daily Packages Delivered Report\n");
            printf("\n");
            printf("9 - Exit The Application\n");
            printf("---------------------------------------------\n");
            Input("Enter your menu choice: ", temp, 2, stdin);
            option = atoi(temp);// Convert char/Str to an integer
            printf("---------------------------------------------\n");

            // Check what choice was entered and act accordingly
            // We can add as many choices as needed
            if( option == 1)
                {
                    Enter_Daily_Packages_Delivered();
                }
            else if(option == 2)
                {
                    Produce_Packages_Delivered_Report();
                }
            else if(option == 9)
                {
                    printf("Exiting the application...\n");
                    Sleep(1000);// Allow 1 second for the Exit notice to display.
                    break;
                }
            else
                {
                    printf("Invalid option.\nPlease enter a number from the Menu Options.\n\n");
                }
        }

    // The following is optional and waits for user input to keep the OS console
    // open before exiting the application.
    system("pause");// Wait until a key is pressed

    return 0;
}// ---> END main application <---
```

```
// ---> START Application Specific Routines
int Enter_Daily_Packages_Delivered()
{
    // Set some variables to hold values for the application
    // Also helps for better code readability
    int min_daily_deliveries = 80;
    int max_daily_deliveries = 170;// Unused ???
    int min_weekly_deliveries = 350;
    int max_weekly_deliveries = 700;
    int good_min_weekly_deliveries = 450;
    int good_max_weekly_deliveries = 600;

    // ---> START Data Structure 1
    // Basic 3 dimensional array to store our data
    // employee_packages_delivered[1][0] = "WeekNumber" (Key)
    // employee_packages_delivered[1][1] = "" (value)
    // employee_packages_delivered[n][n][64] MAX length of string is 64
    char employee_packages_delivered[8][2][INPUTSIZE] =
    {
        {{"WeekNumber"}, {'\0'}},
        {{"EmployeeID"}, {'\0'}},
        {{"EmployeeName"}, {'\0'}},
        {{"Monday"}, {'\0'}},
        {{"Tuesday"}, {'\0'}},
        {{"Wednesday"}, {'\0'}},
        {{"Thursday"}, {'\0'}},
        {{"Friday"}, {'\0'}}
    };

    // Creat a Key/Value look up table structure for employee_packages_delivered
    typedef struct index
    {
        int WeekNumber;
        int EmployeeID;
        int EmployeeName;
        int Monday;
        int Tuesday;
        int Wednesday;
        int Thursday;
        int Friday;
    } index;

    // Populate the look up table so we can use the Key to return
    // and use an integer to locate our data in the 3D array
    // employee_packages_delivered[key."Key"][value]
    struct index key_epd =
    {
        .WeekNumber = 0,
        .EmployeeID = 1,
        .EmployeeName = 2,
        .Monday = 3,
        .Tuesday = 4,
        .Wednesday = 5,
        .Thursday = 6,
        .Friday = 7
    };

    // Record the length of the 3D array for enumeration.
    int Length_employee_packages_delivered = 8;
    // END Data Structure 1 <---

    // ---> START Data Structure 2
    // 3D array to hold Weekly Report.
    char weekly_report[3][2][INPUTSIZE] =
    {
        {{"Employee_1"}, {'\0'}},
        {{"Employee_2"}, {'\0'}},
        {{"Employee_3"}, {'\0'}},
    };

    // Create a key_wr/Value look up table structure for weekly_report
    typedef struct index_wr
    {
```

```
        int Employee_1;
        int Employee_2;
        int Employee_3;
    } index_wr;

    struct index_wr key_wr =
    {
        .Employee_1 = 1,
        .Employee_2 = 2,
        .Employee_3 = 3
    };
    // weekly_report[key."Key"][value]
    int Length_weekly_report = 3;
    // END Data Structure 2 <---

    // key/value used to access both data structures.
    // employee_packages_delivered[n/key_epd."Key"][key/value]
    // [always an int and holds no values][always a char and holds a key and value]
    // [key_epd."Key"][] is used to access the [key_epd."Key"][value]
    // key also be used to access the key name [n][key]
    // Value will always be the 2nd element of the array
    int key = 0;// [n][key]
    int value = 1;// [key_epd."Key"][value]

    char char_Input_Buffer[INPUTSIZE] = {'\0'};// Temp buffer for inputs.
    char char_Temp_Buffer[INPUTSIZE] = {'\0'};// Temp buffer for input manipulations.
    char string_Temp_Buffer[STRINGSIZE] = {'\0'};// 512 characters long. 64 * 8 = 512

    // employee_packages_delivered data structure:
    // 0 WeekNumber, 1 EmployeeID, and 2 EmployeeName are range 0-2
    // Days of week are range 3-7
    // Using these ranges we can loop between employee detail and day entries
    // See routine: Part A) Enter Daily Packages Delivered, and D) Summary for Employee Week
    int starting_day = 3;

// Part A) Enter Employee Details
    // --> Start of For loop for 3 Employees
    int employee_count;
    for(employee_count = 0; employee_count < Length_weekly_report; employee_count++)
        {
            printf("\n");
            printf("\n");
            printf("=============================================\n");
            printf("Enter details for Employee %d\n", (employee_count + 1));
            // employee_count variable starts at 0, so we + 1 to offset the printf
            // employee number message.
            printf("=============================================\n");
            Input("Enter the current working week number >> ", char_Input_Buffer, 3, stdin);
            // Use a temp buffer to build the string "week n"
            // Create part one of the string...
            strcpy(char_Temp_Buffer, "week ");
            // Join our input value to the end of the string.
            strcat(char_Temp_Buffer, char_Input_Buffer);
            // Copy the created string to our array.
            strcpy(employee_packages_delivered[key_epd.WeekNumber][value], char_Temp_Buffer);
            printf("\n");  // Line Break
            Input("Enter the Employee ID >> ", char_Input_Buffer, INPUTSIZE, stdin);
            // Copy the input direct to the array.
            strcpy(employee_packages_delivered[key_epd.EmployeeID][value], char_Input_Buffer);
            printf("\n");  // Line Break
            Input("Enter the employee name >> ", char_Input_Buffer, INPUTSIZE, stdin);
            // Copy the iput direct to the array.
            strcpy(employee_packages_delivered[key_epd.EmployeeName][value], char_Input_Buffer);
            printf("---------------------------------------------\n");
            printf("\n");  // Line Break

// Part A) Enter packages delivered each day
            printf("=============================================\n");
            printf("Enter packages per day for employee %s\n",
employee_packages_delivered[key_epd.EmployeeName][value]);
            printf("=============================================\n");

            int count;
            for(count = 0; count < Length_employee_packages_delivered; count++)
```

```
                {
                    if(count >= starting_day)// element 3
                        {
                            // Monday ->
                            // Note: count = Day
                            printf("Enter employee packages delivered for %s >> ",
employee_packages_delivered[count][key]);// Day == count
                            Input("", char_Input_Buffer, INPUTSIZE, stdin);
                            strcpy(employee_packages_delivered[count][value], char_Input_Buffer);
                            printf("\n");  // Line Break
                        }
                }
            printf("\n-----------------------------------------------\n");
            printf("\n");  // Line Break

// Part B) and D) Summary for Employee Week
            // Build our string week_ID_name_for_heading (String concatenation)
            // "Summary for employee ID:%s NAME:%s Week:%s"
            // 'Summary for employee'" ID:"43" NAME:"Jackson" Week:"17
            char week_ID_name_for_heading[128] = {'\0'};  // A temporary Buffer to hold and manipulate
values.
            strcat(week_ID_name_for_heading, " ID:");
            strcat(week_ID_name_for_heading, employee_packages_delivered[key_epd.EmployeeID][value]);
            strcat(week_ID_name_for_heading, " NAME:");
            strcat(week_ID_name_for_heading,
employee_packages_delivered[key_epd.EmployeeName][value]);
            strcat(week_ID_name_for_heading, " ");// Week:
            strcat(week_ID_name_for_heading, employee_packages_delivered[key_epd.WeekNumber][value]);


// Part B) and D)
            printf("===============================================\n");
            printf("Summary for employee%s\n", week_ID_name_for_heading); // DEBUG created in Part B,
D
            printf("===============================================\n");
            int day_within_limits_flag = 0;
            // A variable to set up a flag that allows for a split if-else block
            // inside of the following loop.
            int total_deliveries = 0;
            for(count = 0; count < Length_employee_packages_delivered; count++)
                {
                    if(count >= starting_day)  // 3 Monday ->
                        {
                            // Add all deliveries for the weekly total.
                            if(atoi(employee_packages_delivered[count][value]) == 0) // error check
                                {
                                    total_deliveries = total_deliveries + 0;
                                }
                            else
                                {
                                    total_deliveries = total_deliveries +
atoi(employee_packages_delivered[count][value]);
                                }
                            //total_deliveries = total_deliveries +
int(employee_packages_delivered[Day])

                            if(atoi(employee_packages_delivered[count][value]) < min_daily_deliveries)
                                {
                                    // See min_daily_deliveries variable at start of program
                                    // where we set this amount.
                                    day_within_limits_flag = 1;
                                    // Flag is set to true, so we can skip the following if that
                                    // is outside of this loop.
                                    printf("%s has not delivered enough packages on %s\n",
employee_packages_delivered[key_epd.EmployeeName][value], employee_packages_delivered[count][key]);
                                }
                            else if(atoi(employee_packages_delivered[count][value]) >
max_daily_deliveries)
                                {
                                    // See max_daily_deliveries variable at start of program
                                    // where we set this amount.
                                    day_within_limits_flag = 1;
                                    printf("%s has delivered too many packages on %s\n",
employee_packages_delivered[key_epd.EmployeeName][value], employee_packages_delivered[count][key]);
```

```
                            }
                        else
                            {
                                // pass
                            }
                    }
                else
                    {
                        // pass
                    }

            }

        if(day_within_limits_flag == 0)
            {
                // The if-else block part 2.  If flag is set to 1 (True) we
                // can skip this.
                printf("%s has delivered within the expected daily packages.\n",
employee_packages_delivered[key_epd.EmployeeName][value]);
            }
        printf("\n%s delivered a total of %d packages in %s\n",
employee_packages_delivered[key_epd.EmployeeName][value], total_deliveries,
employee_packages_delivered[key_epd.WeekNumber][value]);

// Part B)
        strcpy(weekly_report[employee_count][value], itoa(total_deliveries, char_Temp_Buffer,
10));// update the weekly report data structure.
        if(total_deliveries < min_weekly_deliveries)
            {
                // See min_weekly_deliveries variable at start of program
                // where we set this amount.
                printf("%s did not deliver enough packages in week %s\n",
employee_packages_delivered[key_epd.EmployeeName][value],
employee_packages_delivered[key_epd.WeekNumber][value]);
            }
        else if(total_deliveries > max_weekly_deliveries)
            {
                // See max_weekly_deliveries variable at start of program
                // where we set this amount.
                printf("%s delivered too many packages in week %s\n",
employee_packages_delivered[key_epd.EmployeeName][value],
employee_packages_delivered[key_epd.WeekNumber][value]);
            }
        else
            {
                printf("%s has delivered the expected weekly packages.\n",
employee_packages_delivered[key_epd.EmployeeName][value]);
            }
        printf("-----------------------------------------------\n");
        printf("\n");  // Line Break

// Part C) Write to CSV
        // Note! The file write is still inside our for loop and appends the data
        // for each employee until the 3 employee records are reached.
        // The file is opened and then closed for each employee in this example.
        FILE * FileOut; // File open handle
        char file_csv[] = "DailyDeliveries_DB.csv";// Input file. // This is the same as char
*file_csv =
        // Output file to write to.  Notice we save not as *.txt but to *.csv
        //==> Open Output file for text append ops.
        FileOut = fopen(file_csv, "a+");
        if(FileOut == NULL)// (!FileIn) alt. Test if file open success.
            {
                printf("ERROR! Cannot open Output file %s\n", file_csv);
                system("pause");// Wait until a key is pressed
                return -1;
            }

        strcpy(string_Temp_Buffer, ""); // reset/clear the buffer

        int count2;
        for(count2 = 0; count2 < Length_employee_packages_delivered; count2++)
            {
                // Join the elements of employee_packages_delivered to a
```

```
                    // single ',' delimited string.
                    // join each element to the end of buffer string
                    strcat(string_Temp_Buffer, employee_packages_delivered[count2][value]);
                    if(count2 < (Length_employee_packages_delivered -1))
                        {
                            strcat(string_Temp_Buffer, ",");// place the csv separator after each
element
                        }
                    else // The last element [8] will skip adding the separator and add a new line
char instead.
                        {
                            // fprintf appends the newline \n
                        }
                }
            fprintf(FileOut, "%s\n", string_Temp_Buffer);// write the buffer of joined elements to the
next line of the open file (append)
            fclose(FileOut);// we must always remember to close the file when finished
        } // End of For loop for 3 Employees <--

// Part E) Employee Weekly Report
    int not_enough_deliveries = 0;
    int too_many_deliveries = 0;
    int good_number_of_deliveries = 0;
    int employee_number_counter_2 = 0;

    for(employee_number_counter_2 = 0; employee_number_counter_2 < Length_weekly_report;
employee_number_counter_2++)
        {
            if(atoi(weekly_report[employee_number_counter_2][value]) < min_weekly_deliveries)
                {
                    // See min_weekly_deliveries variable at start of program
                    // where we set this amount.
                    not_enough_deliveries = not_enough_deliveries + 1;
                }
            else if(atoi(weekly_report[employee_number_counter_2][value]) > max_weekly_deliveries)
                {
                    // See max_weekly_deliveries variable at start of program
                    // where we set this amount.
                    too_many_deliveries = too_many_deliveries + 1;
                }
            else if((atoi(weekly_report[employee_number_counter_2][value]) >
good_min_weekly_deliveries) && (atoi(weekly_report[employee_number_counter_2][value]) <
good_max_weekly_deliveries))
                {
                    // Axle: I will correct this later.
                    // Python recommends max. 99 characters to one line, so split this
                    // statement across three lines
                    // see https://www.python.org/dev/peps/pep-0008/#maximum-line-length
                    good_number_of_deliveries = good_number_of_deliveries + 1;
                }
            else
                {
                    // pass
                }
        }

// Part E)
    printf("================================================\n");
    printf("Weekly Employee Report\n");
    printf("================================================\n");
    printf("%d employees delivered less than 350 packages a week\n", not_enough_deliveries);
    printf("%d employees delivered more than 700 packages a week\n", too_many_deliveries);
    printf("%d employees delivered between 450-600 packages a week\n", good_number_of_deliveries);
    printf("------------------------------------------------\n");
    printf("\n");  // Line Break
    printf("Press [Enter] to return to the MAIN MENU...\n");
    system("pause");// Wait until a key is pressed.

    return 0;
}// END of Enter_Daily_Packages_Delivered <---

// ================================================================================
// https://www.programiz.com/c-programming/c-dynamic-memory-allocation
// https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/
```

```
// On the "Heap".
// ============================================================================
int Produce_Packages_Delivered_Report()
{
// Part G)
    // fields = ['Week Number', 'Employee ID', 'Employee Name', 'Monday Hrs',
    // 'Tuesday  Hrs', 'Wednesday Hrs', 'Thursday Hrs', 'Friday Hrs']

    FILE *FileIn;// File open handle
    char file_csv[] = "DailyDeliveries_DB.csv";// Input file.
    // Our earlier CSV formatted file. Notice we open *.csv and not *.txt.
    // Build our dynamic 2D/3D array here.
    // In C we also have to allocate enough char space to hold the string values to be stored.
    // Because we don't know in advance how large the csv file will be,
    // we also have to create a dynamic array in memory "On the heap" at run time.

    char char_Temp_Buffer[INPUTSIZE] = {'\0'};// Temp buffer for string manipulations
    char string_Temp_Buffer[STRINGSIZE] = {'\0'};// 512 characters long. 64 * 8 = 512

    int char_buffer;// Temporary buffer to walk the file and count the '\n' newline characters.
    int row_len = 0;// = lines in the text/csv file.
    int col_len = 8;// the 8 elements of employee_packages_delivered.
    int string = 64;// the buffer length to hold the original input data.
    int cnt1, cnt2, cnt3;

    // It is possible that the file may not yet exist. Opening it
    // as "r" will return an exception. Let's test if the file exists first.
    FileIn = fopen(file_csv, "r");// Open file for read ops
    if(FileIn == NULL)//(!FileIn) alt. Test if file open success.
        {
            printf("Error in opening Data file : %s\n", file_csv);
            printf("Maybe the CSV file has not yet been created.\n");
            printf("Please select Option 1 from the MAIN Menu\n");
            printf("to start the data entry.\n");
            printf("Press [Enter] to return to the MAIN MENU...\n");
            system("pause");
            return 0;
        }
    else// Continue to process csv file...
        {
            // For obtaining a byte count. aka number of characters in a file.
            fseek(FileIn, 0, SEEK_END);// Set pointer to end of file.
            int chars_Total = ftell(FileIn);// get counter value.
            rewind(FileIn);// Set pointer back to the start of the file.

            //Read character by character and check for new line
            // I am testing every character in the csv file rather than testing line by line.
            int cnt_chr;
            for(cnt_chr = 0; cnt_chr < chars_Total; cnt_chr++)
                {
                    char_buffer = fgetc(FileIn);

                    if(char_buffer == '\n')// Test if we have encountered a new line and,
                        {
                            row_len++;// increment the number of new lines (row_len) in the file.
                        }
                }
            rewind(FileIn);// Set pointer to start of file (start the next file read from the first
character).

            // Now that we now how many lines/row_len to allocate, we can create a suitable sized
array to hold the contents.
            // We already know the number of columns, and the MAX length of the values.
            // char csv_list_Buffer[lines/row_len][col_len][INPUTSIZE]// columns =
Length_employee_packages_delivered,INPUTSIZE = 64
            // This is somewhat advanced and beyond the scope of a beginner, but I have no other safe
option other than to create
            // the array using pointer arithmetic and dynamic memory. There are multiple ways to
achieve this beyond what I have shown here.
            char ***csv_list_Buffer;// Array to hold csv file read.
            if((csv_list_Buffer = malloc(row_len * sizeof(char **))) != NULL)
                {
                    for(cnt1=0; cnt1 < row_len; cnt1++)
                        {
```

```
                            if((csv_list_Buffer[cnt1]=malloc(col_len * sizeof(char*))) != NULL)
                                {
                                    for(cnt2=0; cnt2 < col_len; cnt2++)
                                        {
                                            if((csv_list_Buffer[cnt1][cnt2]=malloc(string *
sizeof(char*))) != NULL)
                                                {
                                                    for(cnt3=0; cnt3 < string; cnt3++)
                                                        {
                                                            csv_list_Buffer[cnt1][cnt2][cnt3] =
'\0';// Initialise the 3D array to nul
                                                        }
                                                }
                                        }
                                }
                        }
                }
            else
                {
                    //This constitutes an application failure from which we must close the
application.
                    // The user should never receive this error! :)
                    printf("Error - unable to allocate required memory for csv table.\n");
                    return -1;
                }

            // Next we need to read each line to a buffer and then split each value to
            // its correct array location, removing the delimiters ',' and New line chars '\n'
            // Clearing the buffer is not usually required. I just do it for safety when doing string
manipulations.
            memset(char_Temp_Buffer, '\0', INPUTSIZE);// Clear the buffers of previous characters
            memset(string_Temp_Buffer, '\0', STRINGSIZE);// Clear the buffers of previous characters

            int cnt_rows = 0;// track array row position
            int cnt_columns = 0;// track array column position

            while(fgets(string_Temp_Buffer, STRINGSIZE, FileIn) != NULL)
                {
                    // Walk each line from the file (returns ',' in the string
                    // with '\n' at the end).
                    // Strip the newline character from the line.
                    string_Temp_Buffer[strcspn(string_Temp_Buffer, "\r\n")] = '\0';// replace newline
char '\n' with '\0'
                    cnt_columns = 0;// reset the column counter to 0

                    // Use the token ',' to split the string into the original values.
                    // get the first token.
                    char* token = strtok(string_Temp_Buffer, ",");

                    // walk through other tokens
                    while( token != NULL )
                        {
                            strcpy(csv_list_Buffer[cnt_rows][cnt_columns], token);// Copy the
delimited token to the csv 3D array column.
                            token = strtok(NULL, ",");
                            cnt_columns++;// Increment the array position for next column (next token)
                        }

                    cnt_rows++; // move to next line/row and repeat.
                }
            // It is important to free up resources as soon as they are no longer required.
            fclose( FileIn );// finished file reads, close the file.

            printf("=============================================\n");
            printf("Packages Delivered Report\n");
            printf("How many reports would you like to display >> ");
            char temp[8] = {'\0'};
            int options;
            Input("", temp, 8, stdin);
            options = atoi(temp);
            printf("\n---------------------------------------------\n");


            // Check if the report number is more than the entries available.
```

```
            int int_rep_number;
            if(row_len < options)// Note! C Arrays are just integer pointers so we can't accurately
test the length at runtime.
                {
                    int_rep_number = row_len;
                }
            else
                {
                    int_rep_number = options;
                }

            // Calculate the position of the last item in the list
            // minus our report number to display.
            // Note! C Arrays are just integer pointers so we can't accurately test the length at
runtime.
            //We have to record the length as an int variable and use it throughout  the application
when required.
            // remember that a list with 5 elements has an index
            // from [0] to [4], thus the -1
            int report_start = row_len - 1;
            int report_stop = row_len - int_rep_number;

            // Walk through the List in reverse order and printf each
            // line(Row) as text.
            // Walk over list rpt_cnt-- step at a time (or in other words, reversed)
            int rpt_cnt, j;
            // Note! C Arrays are just integer pointers so we can't accurately test the length at
runtime.
            for(rpt_cnt = report_start; rpt_cnt >= report_stop; rpt_cnt--)
                {
                    // csv_list_Buffer steps backward through the number of rows.
                    for(j = 0; j < col_len; j++)
                        {
                            // Step through each element(column) in the row in forward direction.
                            printf("%s\t", csv_list_Buffer[rpt_cnt][j]);// '\t' = TAB, "%s " = single
space
                            // printf each cell value in the row. This will repeat for
                            // the number of col_len in j.
                        }
                    printf("\n");  // End of row [j] col_len, next row/line
                }

            printf("Press [Enter] to return to the MAIN MENU...\n");
            system("pause");

            // Important to always "free" the memory as soon as we are finished with it.
            // Not doing so will lead to a memory leak as a new block of memory will be
            // created on the heap each time the 3D array is used.
            free(csv_list_Buffer);
        }// END file open if, else test.

    return 0;
}
// ---> END Application Specific Routines <---

// ---> START Helper routines

// A wrapper to simplify the fgets() function,
// string formatting and to emulate the Python
// Input() Function
char *Input(char *str, char *buf, int n, FILE *stream)
{
    char *empty = "";
    int ret;
    memset(buf, 0, n);
    ret = strcmp(str, empty);
    if(ret != 0)// Don't print an empty string.
    {
        printf("%s", str);// Input Message
    }
    fseek(stdin, 0, SEEK_END);
    if(fgets (buf, n, stream) != NULL)// if(fgets(buf, sizeof(buf), stdin))
        {
            buf[strcspn(buf, "\r\n")] = 0;// remove end of line
```

```
        }
    return buf;
}

// END Helper routines <---

// ---> Script Exit <---
//------------------------------------------------------------
// NOTES/TODO:
// I Keep the extra notes block here as I often drop unused or temporary working
// out down here until I am finished.
//
//
//
//------------------------------------------------------------
```

Language: FreeBASIC

Code example: “Pract_Task.bas”

```
'################################################################
' Name:          Practical task
' Purpose:       Employee package delivery tracker
' Requires:      See individual Modules following
' Author:        Axle, Daniel
' Contributors:  Add name(s) for anyone who might have helped here
' Copyright:     Add copyright info here, if any, such as
'                (c) Axle 2021, Daniel 2021
' Licence:       Add license here you'd like to use, such as creative
'                commons, GPL, Mozilla Public License, etc.
'                e.g. MIT
'                https://creativecommons.org/licenses/by/2.0/au/
' Created:       07/09/2021 < add the date you start your
'                python file
' Last Modified: 20/10/2021 < add the last date you updated your
'                BASIC source file
' Versioning    ("MAJOR.MINOR.PATCH") Such as Version 1.0.0
'################################################################
' NOTES:
' FreeBASIC is a modernised version of MS QBASIC, Quick BASIC
' It was created to re-imagine old games on modern hardware.
' https://sourceforge.net/projects/fbeginner/files/
' https://versaweb.dl.sourceforge.net/project/fbeginner/Fbguide%20ch%201-7.pdf
' https://www.freebasic.net/wiki/CommunityTutorials
' ^ an excellent resource to check as you code
'################################################################
'----------------------------------------------------------------
' NOTES:
' Updated 19/05/2022
' Example csv database.
'
' TODO: Error handling and string safe routines.
' TODO: Create a wrapper for a clean user Input method.
' TODO: split statements exceeding 80 character width.
'
'----------------------------------------------------------------

' Declare functions used at the bottom of the page (after the formal entry point).
' The interpreter or compiler must read and know all Functions and Subroutines
' that exist before entering the application main routine. Library function, as
' well as our program functions must be read first at the top of the page
' before entering the application. We can let the Interpreter/compiler know
' they exist by declaring the Function at the top of the page, and moving the
' actual function routine to the bottom of the page.
Declare Function main_procedure() As Integer
Declare Function Enter_Daily_Packages_Delivered() As Integer
Declare Function Produce_Packages_Delivered_Report() As Integer

Declare Sub Clear_Stdin()

Main_Procedure() ' Call to the formal application entry
```

```
' ---> START main application
Function Main_Procedure() As Integer ' Formal application entry

    while(1)
        ' Forever loop. Needs a break, return, or exit() statement to
        ' exit the loop.FreeBASIC = Exit While
        ' Placing the menu inside of the while loop "Refreshes" the menu
        ' if an incorrect value is supplied.
        '
        ' Clearing the keyboard buffer is me being pedantic :)
        ' FB uses the GC Compiler and MinGW(32) which can leave
        ' stray '\r''\n' in the stdin buffer.
        Clear_Stdin() ' Clear the keyboard buffer
        Dim options As Integer  ' Declares a menu variable as empty.

        Print "============================================="
        Print "MAIN MENU"
        Print "============================================="
        Print "1 - Enter Daily Packages Delivered"
        Print "2 - Produce Daily Packages Delivered Report"
        Print ""
        Print "9 - Exit The Application"
        Print "---------------------------------------------"
        Clear_Stdin()
        Input ; "Enter your menu choice: ", options
        Print "" ' Line space
        Print "---------------------------------------------"
        ' Print !"\n---------------------------------------------" ' Alternative

        ' Check what choice was entered and act accordingly
        ' We can add as many choices as needed
        If options = 1 Then
            Enter_Daily_Packages_Delivered()
        ElseIf options = 2 Then
            Produce_Packages_Delivered_Report()
        ElseIf options = 9 Then
            Print "Exiting the application..."
            Sleep 1000 ' Allow 1 second for the Exit notice to display.
            ' Alternative: Shell "pause"
            Exit While
        else
            Print !"Invalid option.\nPlease enter a number from the Menu Options.\n"
        End If
    Wend

    ' The following is optional and waits for user input to keep the OS console
    ' open before exiting the application.
    Shell "pause" 'Wait until a key is pressed

    return 0
End Function' ---> END Main_Procedure <---

' ---> START Application Specific Routines
Function Enter_Daily_Packages_Delivered() As Integer

    ' Set some variables to hold values for the application
    ' Also helps for better code readability
    Dim As Integer min_daily_deliveries = 80
    Dim As Integer max_daily_deliveries = 170 ' Unused ???
    Dim As Integer min_weekly_deliveries = 350
    Dim As Integer max_weekly_deliveries = 700
    Dim As Integer good_min_weekly_deliveries = 450
    Dim As Integer good_max_weekly_deliveries = 600

    ' ---> START Data Structure 1
    ' Basic 2 dimensional array to store our data
    ' employee_packages_delivered(1, 0) = "WeekNumber" (Key)
    ' employee_packages_delivered(1, 1) = "" (value)
    Dim employee_packages_delivered(7, 1) As String = {_
    {"WeekNumber", ""},_
    {"EmployeeID", ""},_
    {"EmployeeName", ""},_
    {"Monday", ""},_
    {"Tuesday", ""},_
```

```
    {"Wednesday", ""},_
    {"Thursday", ""},_
    {"Friday", ""}}

    ' Create a Key/Value lookup table structure for employee_packages_delivered
    Type index
        WeekNumber As Integer
        EmployeeID As Integer
        EmployeeName As Integer
        Monday As Integer
        Tuesday As Integer
        Wednesday As Integer
        Thursday As Integer
        Friday As Integer
    End Type

    ' Populate the lookup table so we can use the Key to return
    ' and use an integer to locate our data in the 3D array
    ' employee_packages_delivered(key."Key", value)
    Dim key_epd as index
    key_epd.WeekNumber = 0
    key_epd.EmployeeID = 1
    key_epd.EmployeeName = 2
    key_epd.Monday = 3
    key_epd.Tuesday = 4
    key_epd.Wednesday = 5
    key_epd.Thursday = 6
    key_epd.Friday = 7

    ' Record the length of the 3D array for enumeration.
    Dim As Integer Length_employee_packages_delivered = 8
    ' END Data Structure 1 <---

    ' ---> START Data Structure 2
    ' 3D array to hold Weekly Report.
    Dim weekly_report(2, 1) As String = {_
    {"Employee_1", ""},_
    {"Employee_2", ""},_
    {"Employee_3", ""}}


    ' Create a key_wr/Value look up table structure for weekly_report
    Type index_wr
        Employee_1 As Integer
        Employee_2 As Integer
        Employee_3 As Integer
    End Type

    Dim key_wr As index_wr
    key_wr.Employee_1 = 1
    key_wr.Employee_2 = 2
    key_wr.Employee_3 = 3

    ' weekly_report(key."Key", value)
    Dim As Integer Length_weekly_report = 3
    ' END Data Structure 2 <---

    ' key/value used to access both data structures.
    ' employee_packages_delivered(n/key_epd."Key", key/value)
    ' (always an int and holds no values, always a char and holds a key and value)
    ' (key_epd."Key", ) is used to access the (key_epd."Key", value)
    ' key also be used to access the key name (n, key)
    ' Value will always be the 2nd element of the array
    Dim As Integer key = 0' (n, key)
    Dim As Integer value = 1' (key_epd."Key", value)

    Dim As String char_Input_Buffer = "" ' Temp buffer for inputs.
    Dim As String char_Temp_Buffer = "" ' Temp buffer for input manipulations.
    Dim As String string_Temp_Buffer = "" ' Temp buffer for string manipulations.

    ' employee_packages_delivered data structure:
    ' 0 WeekNumber, 1 EmployeeID, and 2 EmployeeName are range 0-2
    ' Days of week are range 3-7
    ' Using these ranges we can loop between employee detail and day entries
```

```
    ' See routine: Part A) Enter Daily Packages Delivered, and D) Summary for Employee Week
    Dim As Integer starting_day = 3

' Part A) Enter Employee Details
    ' --> Start of For loop for 3 Employees
    Dim employee_count As Integer
    For employee_count = 0 To Length_weekly_report -1 Step 1
        Print ""
        Print ""
        Print "============================================="
        Print "Enter details for Employee "; employee_count + 1
        ' employee_count variable starts at 0, so we + 1 to offset the Print
        ' employee number message.
        Print "============================================="
        Clear_Stdin()
        Input "Enter the current working week number >> ", char_Input_Buffer
        ' Use a temp buffer to build the string "week n"
        ' Create part one of the string and join our input value to the end of the string.
        char_Temp_Buffer = "week " & char_Input_Buffer
        ' Copy the created string to our array.
        employee_packages_delivered(key_epd.WeekNumber, value) = char_Temp_Buffer
        Print "" ' Line Break
        Clear_Stdin()
        Input "Enter the Employee ID >> ", char_Input_Buffer
        ' Copy the input direct to the array.
        employee_packages_delivered(key_epd.EmployeeID, value) = char_Input_Buffer
        Print "" ' Line Break
        Clear_Stdin()
        Input "Enter the employee name >> ", char_Input_Buffer
        ' Copy the input direct to the array.
        employee_packages_delivered(key_epd.EmployeeName, value) = char_Input_Buffer
        Print "---------------------------------------------"
        Print "" ' Line Break

' Part A) Enter packages delivered each day
        Print "============================================="
        Print "Enter packages per day for employee ";
employee_packages_delivered(key_epd.EmployeeName, value)
        Print "============================================="

        Dim count As Integer
        For count = 0 To Length_employee_packages_delivered -1 Step 1
            If count >= starting_day Then' element 3
                ' Monday ->
                ' Note: count = Day
                Clear_Stdin()
                Print "Enter employee packages delivered for "; employee_packages_delivered(count,
key); " >> ";
                Input "", char_Input_Buffer
                employee_packages_delivered(count, value) = char_Input_Buffer
                Print "" ' Line Break
            End If
        Next count

        Print !"\n---------------------------------------------"
        Print "" ' Line Break

' Part B) and D) Summary for Employee Week
        ' Build our string week_ID_name_for_heading (String concatenation)
        ' "Summary for employee ID:var NAME:var Week:var"
        ' 'Summary for employee'" ID:"43" NAME:"Jackson" Week:"17
        Dim week_ID_name_for_heading As String = _ ' A temporary Buffer to hold and manipulate values.
        " ID:" &_' Use & instead of + to include Integer values.
        employee_packages_delivered(key_epd.EmployeeID, value) &_
        " NAME:" &_
        employee_packages_delivered(key_epd.EmployeeName, value) &_
        " " &_ ' Week:
        employee_packages_delivered(key_epd.WeekNumber, value)

' Part B) and D)
        'https://www.freebasic.net/forum/viewtopic.php?t=1626 ' atoi()
        Print "============================================="
        Print "Summary for employee "; week_ID_name_for_heading ' DEBUG created in Part B, D
        Print "============================================="
```

```
        Dim As Integer day_within_limits_flag = 0
        ' A variable to set up a flag that allows for a split if-else block
        ' inside of the following loop
        Dim As Integer total_deliveries = 0
        For count = 0 To Length_employee_packages_delivered -1 Step +1
            If count >= starting_day Then' 3 Monday ->
                ' Add all deliveries for the weekly total.
                If ValInt(employee_packages_delivered(count, value)) = 0 Then ' error check
                    total_deliveries = total_deliveries + 0
                Else
                    total_deliveries = total_deliveries + ValInt(employee_packages_delivered(count,
value))
                End If

                If ValInt(employee_packages_delivered(count, value)) < min_daily_deliveries Then
                    ' See min_daily_deliveries variable at start of program
                    ' where we set this amount.
                    day_within_limits_flag = 1
                    ' Flag is set to true, so we can skip the following if that
                    ' is outside of this loop.
                    Print employee_packages_delivered(key_epd.EmployeeName, value);" has not delivered
enough packages on "; employee_packages_delivered(count, key)
                ElseIf ValInt(employee_packages_delivered(count, value)) > max_daily_deliveries Then
                    ' See max_daily_deliveries variable at start of program
                    ' where we set this amount.
                    day_within_limits_flag = 1
                    Print employee_packages_delivered(key_epd.EmployeeName, value);" has delivered too
many packages on ";employee_packages_delivered(count, key)
                End If
            End If

        Next count

        If day_within_limits_flag = 0 Then
            ' The if-else block part 2.  If flag is set to 1 (True) we
            ' can skip this.
            Print employee_packages_delivered(key_epd.EmployeeName, value);" has delivered within the
expected daily packages."
        End If
        Print employee_packages_delivered(key_epd.EmployeeName, value);" delivered a total
of";total_deliveries;" packages in ";employee_packages_delivered(key_epd.WeekNumber, value)

' Part B)
        weekly_report(employee_count, value) = Str(total_deliveries)' update the weekly report data
structure.
        If total_deliveries < min_weekly_deliveries Then
            ' See min_weekly_deliveries variable at start of program
            ' where we set this amount.
            Print employee_packages_delivered(key_epd.EmployeeName, value);" did not deliver enough
packages in week ";employee_packages_delivered(key_epd.WeekNumber, value)
        ElseIf total_deliveries > max_weekly_deliveries Then
            ' See max_weekly_deliveries variable at start of program
            ' where we set this amount.
            Print employee_packages_delivered(key_epd.EmployeeName, value);" delivered too many
packages in week ";employee_packages_delivered(key_epd.WeekNumber, value)
        Else
            Print employee_packages_delivered(key_epd.EmployeeName, value);" has delivered the
expected weekly packages."
        End If

        Print "-----------------------------------------------"
        Print ""  ' Line Break

' Part C) Write to CSV
        ' Note! The file write is still inside our for loop and appends the data
        ' for each employee until the 3 employee records are reached.
        ' The file is opened and then closed for each employee in this example.
        Const file_csv As String = "DailyDeliveries_DB.csv" ' Input file.
        ' Output file to write to.  Notice we save not as *.txt but to *.csv
        '==> Open Output file for text append ops.
        Dim FileOut As Integer = FreeFile()
        If 0 <> Open(file_csv, For Append, As FileOut) Then
            Print "ERROR! Cannot open Output file " & file_csv
            Shell "pause"' Wait until a key is pressed
```

```
            Return -1
        End If

        string_Temp_Buffer = "" ' reset/clear the buffer

        Dim count2 As Integer
        For count2 = 0 To Length_employee_packages_delivered -1 Step 1
            ' Join the elements of employee_packages_delivered to a
            ' single ',' delimited string.
            ' join each element to the end of buffer string
            string_Temp_Buffer = string_Temp_Buffer & employee_packages_delivered(count2, value)
            If count2 < Length_employee_packages_delivered -1 Then
                string_Temp_Buffer = string_Temp_Buffer & ","' place the csv separator after each
element
            Else ' The last element [8] will skip adding the separator and add a new line char
instead.
                ' Print # appends the newline \n
            End If
        Next count2
        ' Write the buffer of joined elements to the next line of the open file (append).
        Print #FileOut, string_Temp_Buffer
        ' We must always remember to close the file when finished.
        Close #FileOut

    Next employee_count' End of For loop for 3 Employees <--

' Part E) Employee Weekly Report
    Dim As Integer not_enough_deliveries = 0
    Dim As Integer too_many_deliveries = 0
    Dim As Integer good_number_of_deliveries = 0
    Dim As Integer employee_number_counter_2 = 0

    for employee_number_counter_2 = 0 To Length_weekly_report-1 Step 1
        If ValInt(weekly_report(employee_number_counter_2, value)) < min_weekly_deliveries Then
            ' See min_weekly_deliveries variable at start of program
            ' where we set this amount.
            not_enough_deliveries = not_enough_deliveries + 1
        ElseIf ValInt(weekly_report(employee_number_counter_2, value)) > max_weekly_deliveries Then
            ' See max_weekly_deliveries variable at start of program
            ' where we set this amount.
            too_many_deliveries = too_many_deliveries + 1
        ElseIf (ValInt(weekly_report(employee_number_counter_2, value)) > good_min_weekly_deliveries)
And (ValInt(weekly_report(employee_number_counter_2, value)) < good_max_weekly_deliveries) Then
            ' Axle: I will correct this later.
            ' Python recommends max. 99 characters to one line, so split this
            ' statement across three lines
            ' see https:'www.python.org/dev/peps/pep-0008/#maximum-line-length
            good_number_of_deliveries = good_number_of_deliveries + 1
        Else
            ' pass
        End If
    Next employee_number_counter_2

' Part E)
    Print "================================================"
    Print "Weekly Employee Report"
    Print "================================================"
    Print not_enough_deliveries;" employees delivered less than 350 packages a week"
    Print too_many_deliveries;" employees delivered more than 700 packages a week"
    Print good_number_of_deliveries;" employees delivered between 450-600 packages a week"
    Print "------------------------------------------------"
    Print ""  ' Line Break
    Print "Press [Enter] to return to the MAIN MENU..."
    Shell "pause"' Wait until a key is pressed.

    return 0
End Function ' END of Enter_Daily_Packages_Delivered <---

' ================================================================================
' https://documentation.help/FreeBASIC/ProPgVarLenArrays.html
'   Dim array(Any)
'   ReDim array(n elements)
' On the "Heap".
' ================================================================================
```

```
Function Produce_Packages_Delivered_Report() As Integer
    '' Note!! An array(9) has 10 elements 0 To 9.
    '' As we don't use counter < length in for loops in basic, we must use length -1
    ''
    '' Dim As Integer length = 5
    '' For y = 0 To length -1 Step 1 ' (aka When y = 5-1)
    ''    0,1,2,3,4
    '' Next y

' Part G)
    ' fields = ['Week Number', 'Employee ID', 'Employee Name', 'Monday Hrs',
    ' 'Tuesday  Hrs', 'Wednesday Hrs', 'Thursday Hrs', 'Friday Hrs']

    Const file_csv As String = "DailyDeliveries_DB.csv"' Input file.
    ' Our earlier CSV formatted file. Notice we open *.csv and not *.txt.
    ' Build our dynamic 2D/3D array here.
    ' in FreeBASIC we also have to allocate enough char space to hold the string values to be stored.
    ' Because we don't know in advance how large the file will be,
    ' we also have to create a dynamic array in memory "On the heap" at run time.

    Dim As String csv_list_Buffer(Any,Any) ' Array declared on dynamic memory (On the heap).

    Dim temp_buffer As String ' Temporary buffer

    Dim As Integer x, y ' To enumerate and set array parameters.
    Dim As Integer row_len = 0 ' = lines in the text/csv file (To be calculated).
    Dim As Integer col_len = 8 ' The 8 elements of employee_packages_delivered.

    Dim FileIn As Integer = Freefile()
    ' It is possible that the file may not yet exist. Opening it
    ' as "read/Input" will return an error. Let's test if the file exists first.
    If Open(file_csv, For Input, As FileIn) <> 0 Then
        Print "ERROR! Cannot open Output file " & file_csv
        Print "Maybe the CSV file has not yet been created."
        Print "Please select Option 1 from the MAIN Menu"
        Print "to start the data entry."
        Print "Press [Enter] to return to the MAIN MENU..."
        Shell "pause"'wait until a key is pressed
        Return 0

    Else' Continue to process csv file...

        ' Check the file for the number of lines/Rows.
        ' Line Input # will read one line (up to '\r''\n') at a time.
        While Not Eof (FileIn)
            Line Input #FileIn, temp_buffer
            row_len = row_len + 1
        Wend
        Seek #FileIn, 1 ' Set pointer back to the start of the file.

        ' Now that we know how many lines/rows to allocate, we can create a suitable sized array to
hold the contents.
        ' We already know the number of columns (= 8).
        ' Using ReDim we can set the required size of the dynamic array on the heap at runtime.
        ' (8 -1 = 7) | 0,1,2,3,4,5,6,7 = 8 elements
        Redim csv_list_Buffer(row_len -1, col_len -1)
        ' Read the file into Data Structure 3 (2 dimensional Dynamic Array)
        ' at its correct array location, removing the delimiters ',' and New line chars '\n'
        ' The following method is designed for CSV files and automatically removes the ',' delimiter
and newline characters.

        For y = 0 To row_len -1
            For x = 0 To col_len -1
                Input #FileIn, csv_list_Buffer(y,x) ' add each value to the correct array position by
(row, column).
            Next x
        Next y

        ' It is important to free up resources as soon as they are no longer required.
        Close #FileIn' finished file reads, close the file.

        Print "============================================="
        Print "Packages Delivered Report"
        Dim  options As Integer
```

```
        Clear_Stdin()
        Input "How many reports would you like to display >> ",  options
        Print !"\n-----------------------------------------------"

        ' Check if the report number is more than the entries available.
        Dim int_rep_number As Integer
        If row_len <  options Then' Note! C Arrays are just integer pointers so we can't accurately
test the length at runtime.
            int_rep_number = row_len
        Else
            int_rep_number =  options
        End If

        ' Calculate the position of the last item in the list
        ' minus our report number to display.
        ' I have recorded the length of Rows and don't have to recalculate it's value
        ' remember that a list with 5 elements has an index
        ' from [0] to [4], thus the -1
        Dim As Integer report_start = row_len
        Dim As Integer report_stop = row_len - int_rep_number

        ' Walk through the List in reverse order and Print each
        ' line(Row) as text.
        ' Walk over list Step -1 step at a time (or in other words, reversed)
        Dim As Integer rpt_cnt, j
        For rpt_cnt = report_start -1 To report_stop Step -1 ' Step Backward (+1 to fix alignment)
            ' csv_list_Buffer steps backward through the number of rows.

            For j = 0 To col_len -1 Step 1
                ' Step through each element(column) in the row in a forward direction.
                Print csv_list_Buffer(rpt_cnt, j);!"\t"; ' '\t' = TAB, "%s " = single space
                ' Print each cell value in the row. This will repeat for
                ' the number of columns in j.
            Next j
            Print ""  ' End of row [j] columns, next row/line
        Next rpt_cnt

        Print "Press [Enter] to return to the MAIN MENU..."
        Shell "pause"'wait until a key is pressed
        'system("pause");

        ' Important to always "free" the memory as soon as we are finished with it.
        ' Not doing so will lead to a memory leak as a new block of memory will be
        ' created on the heap each time the dynamic array is used.
        ' I am uncertain if dynamic memory (On the Heap) is freed when the Function exits,
        ' so I am just playing safe ¯\_(''/)_/¯
        Erase csv_list_Buffer
    End If' END file open if, else test.

    Return 0
End Function
' ---> END Application Specific Routines <---

' ---> START Helper routines

' A wrapper to flush/clear the keyboard input buffer
Sub Clear_Stdin()
    While Inkey <> ""  '' loop until the Inkey buffer is empty
    Wend
End Sub

' END Helper routines <---

' ---> Script Exit <---
'-------------------------------------------------------------
' NOTES/TODO:
' I Keep the extra notes block here as I often drop unused or temporary working
' out down here until I am finished.
'
'
'
'-------------------------------------------------------------
```

Language: Python 3

Code example: “Pract_Task.py”

```
#############################################################
# Name:          Practical task
# Purpose:       Employee package delivery tracker
# Requires:      See individual Modules following
# Author:        Axle, Daniel
# Contributors:  Add name(s) for anyone who might have helped here
# Copyright:     Add copyright info here, if any, such as
#                (c) Axle 2021, Daniel 2021
# Licence:       Add license here you'd like to use, such as creative
#                commons, GPL, Mozilla Public License, etc.
#                e.g. MIT
#                https://creativecommons.org/licenses/by/2.0/au/
# Created:       07/09/2021 < add the date you start your
#                python file
# Last Modified: 20/10/2021 < add the last date you updated your
#                python source file
# Versioning    ("MAJOR.MINOR.PATCH") Such as Version 1.0.0
#############################################################
# NOTES:
# https://www.w3schools.com/python/default.asp
# ^ an excellent resource to check as you code
#############################################################
#-------------------------------------------------------------
# NOTES:
# Updated 06/01/2022
# Example csv database.
#
# TODO: Error handling and string safe routines.
# TODO!!! Fix indentation ALL
# TODO: split statements exceeding 80 character width.
#
#-------------------------------------------------------------

# I am only importing the sleep function from the time library
from time import sleep

# The interpreter or compiler must read and know all Functions and Subroutines
# that exist before entering the application main routine. Library function, as
# well as our program functions must be read first at the top of the page
# before entering the application. In Python the formal entry point is the
# first If statement, so we place
# if __name__ == '__main__':
#     main()
# as the last line to force the Python interpreter to read the entire page
# before starting the application main()

def main():  # Formal application entry after first IF

    while(True):
        # Forever loop. Needs a break, return, or exit() statement to
        # exit the loop.
        # Placing the menu inside of the while loop "Refreshes" the menu
        # if an incorrect value is supplied.
        # Notice the Multiline """ String

        option = None  # Declares a menu variable as empty.

        print("""=============================================
MAIN MENU
=============================================
1 - Enter Daily Packages Delivered
2 - Produce Daily Packages Delivered Report

9 - Exit The Application
---------------------------------------------""")
        options = input("Enter your menu choice: ")
        print("---------------------------------------------")

        # Check what choice was entered and act accordingly
        # We can add as many choices as needed
```

```
        if options == '1':
            Enter_Daily_Packages_Delivered()
        elif options == '2':
            Produce_Packages_Delivered_Report()
        elif options == '9':
            print("Exiting the application...")
            sleep(1)  # Allow 1 second for the Exit notice to display.
            break
        else:
            print("Invalid option.\nPlease enter a number from the Menu Options.\n")

    # The following is optional and waits for user input to keep the OS console
    # open before exiting the application.
    input("Press [Enter] to exit.")

    # Returns to the line directly after the call to main()
    # at the bottom of the page.
    return None
## ---> END of MAIN MENU <---

def Enter_Daily_Packages_Delivered():
    # Set some variables to hold values for the application
    # Also helps for better code readability
    min_daily_deliveries = 80
    max_daily_deliveries = 170
    min_weekly_deliveries = 350
    max_weekly_deliveries = 700
    good_min_weekly_deliveries = 450
    good_max_weekly_deliveries = 600

    # Data Structure 1
    # https://www.w3schools.com/python/python_dictionaries.asp
    employee_packages_delivered = {
    "WeekNumber": "",
    "EmployeeID": "",
    "EmployeeName": "",
    "Monday": "",
    "Tuesday": "",
    "Wednesday": "",
    "Thursday": "",
    "Friday": "",
    }

    # employee_packages_delivered data structure:
    # 0 WeekNumber, 1 EmployeeID, and 2 EmployeeName are range 0-2
    # Days of week are range 3-7
    # Using these ranges we can loop between employee detail and day entries
    # See routine: Part A) Enter Daily Packages Delivered, and D) Summary for Employee Week

    starting_day = 3

    # Data Structure 2
    weekly_report = {
    "Employee 1": 0,
    "Employee 2": 0,
    "Employee 3": 0,
    }

# Part A) Enter Employee Details
    # Start of For loop for 3 Employees
    for employee_count, employee_number in enumerate(weekly_report):
        print()
        print()
        print("================================================")
        print("Enter details for Employee", employee_count + 1)
        # employee count variable starts at 0, so we + 1 to offset the print
        # employee number message.
        print("================================================")
        employee_packages_delivered["WeekNumber"] = "week " + str(input("Enter the current
working week number >> "))
        print()  # Line Break
        employee_packages_delivered["EmployeeID"] = str(input("Enter the Employee ID >> "))
        print()  # Line Break
        employee_packages_delivered["EmployeeName"] = str(input("Enter the employee name >>
"))
        print("------------------------------------------------")
```

```
        print()  # Line Break

# Part A) Enter packages delivered each day
        print("================================================")
        print("Enter packages per day for employee ",
employee_packages_delivered["EmployeeName"])
        print("================================================")
        # https://www.w3schools.com/python/python_dictionaries.asp
        # https://www.programiz.com/python-programming/methods/built-in/enumerate
        for count, Day in enumerate(employee_packages_delivered):
            if  count >= starting_day:  # Monday ->
                print("Enter employee packages delivered for ", Day, " >> ", end='')
                employee_packages_delivered[Day] = str(input())
                print()  # Line Break
        print("------------------------------------------------")
        print()  # Line Break

# Part B) and D) Summary for Employee Week
        # Build our string week_ID_name_for_heading (String concatenation)
        # "Summary for employee ID:%s NAME:%s Week:%s"
        # 'Summary for employee'" ID:"43" NAME:"Jackson" Week:"17
        week_ID_name_for_heading = "ID:"
        week_ID_name_for_heading = week_ID_name_for_heading +
employee_packages_delivered["EmployeeID"]
        week_ID_name_for_heading = week_ID_name_for_heading + " NAME:"
        week_ID_name_for_heading = week_ID_name_for_heading +
employee_packages_delivered["EmployeeName"]
        week_ID_name_for_heading = week_ID_name_for_heading + " "
        week_ID_name_for_heading = week_ID_name_for_heading +
employee_packages_delivered["WeekNumber"]

# Part B) and D)
        print("================================================")
        print("Summary for employee", week_ID_name_for_heading)
        print("================================================")
        day_within_limits_flag = 0
        # A variable to set up a flag that allows for a split if-else block
        # inside of the following loop.
        total_deliveries = 0
        for count, Day in enumerate(employee_packages_delivered):
            if  count >= starting_day:  # Monday ->
                # Add all deliveries for the weekly total.
                total_deliveries = total_deliveries + int(employee_packages_delivered[Day])
                if int(employee_packages_delivered[Day]) < min_daily_deliveries:
                    # See min daily deliveries variable at start of program
                    # where we set this amount.
                    day_within_limits_flag = 1
                    # Flag is set to true, so we can skip the following if that
                    # is outside of this loop.
                    print(employee_packages_delivered["EmployeeName"], "has not delivered
enough packages on", Day)
                elif int(employee_packages_delivered[Day]) > max_daily_deliveries:
                    # See max_daily_deliveries variable at start of program
                    # where we set this amount.
                    day_within_limits_flag = 1
                    print(employee_packages_delivered["EmployeeName"], "has delivered too many
packages on", Day)
            else:
                pass
        if day_within_limits_flag == 0:
            # The if-else block part 2.  If flag is set to 1 (True) we
            # can skip this.
            print(employee_packages_delivered["EmployeeName"], "has delivered within the
expected daily packages.")

        print(employee_packages_delivered["EmployeeName"], "delivered a total of",
total_deliveries, "packages in week", employee_packages_delivered["WeekNumber"])

# Part B)
        weekly_report[employee_number] = total_deliveries
        if total_deliveries < min_weekly_deliveries:
            # See min_weekly_deliveries variable at start of program
            # where we set this amount.
            print(employee_packages_delivered["EmployeeName"], "did not deliver enough
packages in week", employee_packages_delivered["WeekNumber"])
        elif total_deliveries > max_weekly_deliveries:
```

```
            # See max_weekly_deliveries variable at start of program
            # where we set this amount.
            print(employee_packages_delivered["EmployeeName"], "delivered too many packages in
week", employee_packages_delivered["WeekNumber"])
        else:
            print(employee_packages_delivered["EmployeeName"], "has delivered the expected
weekly packages.")

        print("-----------------------------------------------")
        print()  # Line Break

# Part C) Write to CSV
        # Note! The file write is still inside our for loop and appends the data
        # for each employee until the 3 employee records are reached.
        # The file is opened and then closed for each employee in this example.
        file_csv_out = "DailyDeliveries_DB.csv"
        # Output file to write to.  Notice we save not as *.txt but to *.csv
        # ==> Open Output file for text append ops
        with open(file_csv_out, "a") as file:
            str_buffer = ",".join(employee_packages_delivered.values())
            # Join the elements of employee_packages_delivered to a
            # single ',' delimited string.
            str_buffer = str_buffer + '\n'  # append a new line character \n
            file.write(str_buffer)
            # Write/append 'a' str_buffer to file.

    # End of For loop for 3 Employees <---

# Part E) Employee Weekly Report
    not_enough_deliveries = 0
    too_many_deliveries = 0
    good_number_of_deliveries = 0

    for employee_number_counter_2 in weekly_report:
        if weekly_report[employee_number_counter_2] < min_weekly_deliveries:
            # See min weekly deliveries variable at start of program
            # where we set this amount.
            not_enough_deliveries = not_enough_deliveries + 1
        elif weekly_report[employee_number_counter_2] > max_weekly_deliveries:
            # See max_weekly_deliveries variable at start of program
            # where we set this amount.
            too_many_deliveries = too_many_deliveries + 1
        elif (weekly_report[employee_number_counter_2]
        > good_min_weekly_deliveries) and (int(weekly_report[employee_number_counter_2])
        < good_max_weekly_deliveries):
            # From earlier elif to here, this could be all on one line
            # Python recommends max. 99 characters to one line, so split this
            # statement across three lines
            # see https://www.python.org/dev/peps/pep-0008/#maximum-line-length
            good_number_of_deliveries = good_number_of_deliveries + 1
        else:
            pass

# Part E)
    print("================================================")
    print("Weekly Employee Report")
    print("================================================")
    print(not_enough_deliveries, "employees delivered less than 350 packages a week")
    print(too_many_deliveries, "employees delivered more than 700 packages a week")
    print(good_number_of_deliveries, "employees delivered between 450-600 packages a week")
    print("------------------------------------------------")
    print()  # Line Break
    input("Press [Enter] to return to the MAIN MENU...")

    return None
# END of Enter_Daily_Packages_Delivered <---

def Produce_Packages_Delivered_Report():
# Part G)

    # fields = ['Week Number', 'Employee ID', 'Employee Name', 'Monday Hrs',
    # 'Tuesday  Hrs', 'Wednesday Hrs', 'Thursday Hrs', 'Friday Hrs']
    # range(start, stop)
    file_csv_in = "DailyDeliveries_DB.csv"
    # Our earlier CSV formatted file. Notice we open *.csv and not *.txt.
    csv_list = []
```

```
    # main Data List. CSV converted to list.

    # It is possible that the file may not yet exist. Opening it
    # as "r" will return an exception. Let's test if the file exists first.
    try:
        with open(file_csv_in, "r") as file:  # Open the CSV File.
        # read the file into Data Structure 3 (2 dimensional Linked List)
            for buffer in file:
                # Walk each line from the file (returns ',' in the string
                # with '\n' at the end
                buffer = buffer.strip('\n')
                # Strip the newline character from the line.
                buffer_list = buffer.split(',')
                # Separate the lines at ',' to a single list.
                csv_list.append(buffer_list)
                # Append each single dimension list to a 2 dimension list.

        print("================================================")
        print("Packages Delivered Report")
        options = input('How many reports would you like to display >> ')
        # Will default to str/text input
        print("------------------------------------------------")

        # Check if the report number is more than the entries available.
        if len(csv_list) < int(options):
            int_rep_number = len(csv_list)
        else:
            int_rep_number = int(options)

        # Calculate the position of the last item in the list
        # minus our report number to display.
        report_start = len(csv_list) - 1
        report_stop = len(csv_list) - int_rep_number - 1
        report_step = -1
        # Walk over list -1 step at a time (or in other words, reversed)

        # Walk through the List in reverse order and print each
        # line(Row) as text.
        # remember that a list with 5 elements has an index
        # from [0] to [4], thus the -1
        # range() automatically converts real numbers to index numbers
        for i in range(report_start, report_stop, report_step):
            # csv_list steps backward through the number of rows.
            for j in range(len(csv_list[i])):
                # Step through each element(column) in the row.
                print(csv_list[i][j], end=" ")
                # print each cell value in the row. This will repeat for
                # the number of columns in j.
            print()  # End of row [j], next i in range()

        input("Press [Enter] to return to the MAIN MENU...")
        return None

    except FileNotFoundError:  # If the file does not yet exist.
        print("The CSV file has not yet been created.")
        print("Please select Option 1 from the MAIN Menu")
        print("to start the data entry.")
        input("Press [Enter] to return to the MAIN MENU...")
        return None

## ---> END Application Specific Routines <---

if __name__ == '__main__':
    main()

## ---> Script Exit <---
#---------------------------------------------------------------
# NOTES/TODO:
# I Keep the extra notes block here as I often drop unused or temporary working
# out down here until I am finished.
#
#
#
#---------------------------------------------------------------
```

Additional components

"… Difficult to master"

Previously I have stated the quote "Easy to learn, difficult to master". Learning the fundamentals of programming, although challenging, is not an overly difficult endeavour. There are many good guides for beginners that will help in getting started with individual languages, although few will target the concept of programming from a language neutral perspective.

Once we have developed confidence in the fundamental programming and coding principles we will want to progress to more advanced programming and coding concepts. This is where I have found a gap in the current educational system where beginners level guides are abundant, engineer level guides are abundant but the bridge between the two appear lacking and can be a major stumbling block for a student.

I am not intending on going into too much detail in this section as a feel that it would require at a minimum a separate booklet covering all of the intermediate programming concepts, but I will attempt to cover at least the most fundamental areas to assist a student in finding the light switches that allow further study and progress. One of the most difficult issues is not knowing what you are looking for until after you have found it.

Below is a brief outline of some of the next steps to at least keep some momentum toward creating sharable production software.

Styles

Coding styles can vary from language to language and over time we have a natural tendency toward developing our own coding style. There are general guides and standards that exist in most programming languages and it is important to make an effort towards following the coding styles that are set out. Styles can be ambiguous and even contradictory when working across multiple languages.

Programming style guides and conventions can be somewhat opinionated and project centric, but there are basic conventions that will remain for most languages. The important thing is to attempt to be consistent especially within the boundaries of an individual coding project.

A lot can be learned from reading the code of professionals. One of the most readily available sources of code is the libraries or modules that are installed alongside the programming language. Most formally recognised library modules are very well written and tend to be standards compliant. Spending a little time in github looking at how others can offer some insight into both good and poor style practices.

Even I can be a bit sloppy with code styling, mostly because I swap around between different languages and and unconsciously mix the styles up.

Some syle guides for reference:

- **C Language**
  https://www.cs.umd.edu/~nelson/classes/resources/cstyleguide/
  https://users.ece.cmu.edu/~eno/coding/CCodingStandard.html
  https://www.doc.ic.ac.uk/lab/cplus/cstyle.html
  https://www.gnu.org/prep/standards/html_node/Writing-C.html
  https://github.com/MaJerle/c-code-style
- **C/C++:**
  https://google.github.io/styleguide/cppguide.html

- **FreeBASIC**
  I could not find a pure "FreeBASIC" guide. MS Visual Basic is similar in syntax as it is derived from QBASIC. FreeBASIC is also syntactically correct QBASIC. We could really place this under the language heading of "BASIC" as most variations of BASIC follow a similar syntax.
  https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/program-structure/program-structure-and-code-conventions

- **PEP 8 – Style Guide for Python Code**
  https://peps.python.org/pep-0008/

Core components of coding style

Comments

Use meaningful comments to explain the expected outcome or purpose of a statement or expression. If the program flow or logic is ambiguous, add a comment to explain. Anyone else should be able to read your code and follow the flow and logic easily.

Include a single white space after the comment symbol

```
// My comment.
```

Include a minimum of 2 spaces before an inline comment

`Print "Hello world"  ' My comment.` Note the 2 spaces …ld"12' My…
It is good practice to line up multiple lines of inline comments at the TAB stops.

```
int MessageBox(
  [in, optional] HWND    hWnd,      // handle to the owner window
  [in, optional] LPCTSTR lpText,    // message to be displayed
  [in, optional] LPCTSTR lpCaption, // dialog box title
  [in]           UINT    uType      // behavior of the dialog box
);
```

Names

The most common style for names is snake case `my_variable`. Use names that are short, meaningful and are consistent within the context of the project environment. Suffixes and

prefixes are commonly used to illustrate the context in which it is used; value_max, value_min, mylib_addition(num_1, num_2).

Global Variables and Constants

Use all caps to define global variables, constants and macros.

```
#define MAX_LEN 10
#define MY_MACRO(x)         ((x) * (x))
const int A_GLOBAL_CONSTANT= 5;
Dim As Integer MAX_LIMIT = 128
DEFAULT_VALUE = 100
```

Magic Numbers

Avoid using magic numbers (aka Numeric literals) throughout an application. Assign it a meaningful variable name so that we can understand its purpose. It is best to place these literals as a set of constants at the start of a function block or routine.

```
// NO. Don't do this.
out = 2*3.14*15.2

// This is more understandable.
PI = 3.14
circumference = 0.0
radius = 15.2
diameter = 2 * radius

circumference = PI * diameter
```

Sometimes it is difficult to avoid numeric literals and it is OK to use them if their purpose is easy to understand.

```
// This is also OK.
PI = 3.14
circumference = 0.0
radius = 15.2

circumference = 2 * PI * radius
```

Indentation

Indentation makes our code readable to humans. Indentation also defines the start and end of statements and expression blocks as well as the level in the child parent relationship. Each indentation to the right signifies that this is a sub-process of the parent process. Use 4 space indentations and do not use TABs. Some IDEs will allow the use of the TAB to create a 4 space indentation and it is important to check the configuration to ensure TABS are always written as 4 spaces rather than an actual TAB.

```
My_function()
      MY_CONST = 5
```

```
        My_var = 0
        While (MY_CONST != My_var)
              If (x == My_var) Then
                    Do something
              Else If (y == My_var) Then
                    Do something
              Else
                    Nothing evaluated to True
              End If
        End While
End Function
```

Parens ()

Leave a space between all statements and conditionals and no space in a function call

```
        My_function()
        MY_CONST = 5
        My_var = 0
        While (MY_CONST != My_var)
              If (x == My_var) Then
                    Print("Hello")
              Else If (y == My_var) Then
                    Print("Goodbye")
              Else
                    Nothing evaluated to True
              End If
        End While
End Function
```

Line Length

Consoles have traditionally been 80 characters wide. As such, most code editors have been derived from TUI text editors and have a default viewing width of 80 characters. There is no enforced length of a line in a text document or even a text editor, but we attempt to stay within the boundary of 80 characters for the ease of readability as this tends to be the default display width, even though we can scroll the editor to the right for long lines.

When reading through source code we need all code to remain within that 80 character view port so all code is readable. If it goes off the page and out of view we can easily miss what we are looking for.

This can be a difficult task and may require some function arguments and statements to be spread over multiple lines. We can do this using a line continuation character, or for some languages we can just split the statement at a meaningful position.

```
int MessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT
uType);

int MessageBox(
  HWND    hWnd,
  LPCTSTR lpText,
  LPCTSTR lpCaption,
```

```
  UINT    uType
);

If ((variable_a == Variable_x && variable_b < variable_y && variable
_c > variable_z) || (variable_a < Variable_z && variable_b >
variable_y && variable _c > variable_x)) Then
        Do Something
End If

If ((variable_a == Variable_x
        && variable_b < variable_y
        && variable _c > variable_z)
        || (variable_a < Variable_z
        && variable_b > variable_y
        && variable _c > variable_x)) Then
        Do Something
End If
```

White space

Use white spaces in a similar way to how you would structure your grammar in a normal document. White space (except for python) is generally ignored by the compiler. Only use white space that is required and don't use excessive white space to separate lines of code. A vertical space is acceptable to separate meaningful blocks of code.

Use spaces rather than TABS, generally 4 spaces.

Whitespace in comments makes it easier to recognize comments when you're scrolling through a codebase. This applies to both inline and separate comments.

Equality

Place the constant to the left side in an equality statement `If (5 == value) Then`

This removes compiler and interpreter confusion and stops us from accidentally assigning a value to a variable instead of testing for equality.

```
If (value = 5) Then  // This will attempt to assign 5 to the variable
value.

If (5 = value) Then  // This will always throw an error as we cannot assign
a value to a literal.
```

Statements

It is best practice to place one statement per line. Some people advocate for the use of "One Liners". This is acceptable for exported production code, or for obfuscated code but not for source code as it makes it difficult to read. That being said, "Functional Programming" does make extensive use of "One Liners", so it really is a balance in keeping a high degree of readability.

Libraries (Modules)

I am only going to offer a brief explanation of using libraries and included modules in this booklet. I will make use of libraries as well as numerous examples in the "A BEGINNERS GUIDE TO PROGRAMMING" books 3, 4 and 5; C/C++ Overview, FreeBASIC Overview and Python 3 Overview.

One of the most important additions are "Libraries". Libraries are large collections of well tested, well proven, reusable code blocks. Libraries are created from the programming language keywords and in some cases even from other libraries.

Instead of writing many thousands of lines of source code every time we start a new project, we can use the premade code from libraries. Libraries (Collections) will come in many categories such as GUI libraries, graphics libraries, math libraries, game creation libraries, and so on. These libraries have been created by many programmers and engineers that have done all of the hard coding work, and most often we just need to add a few lines of code to call complex procedures from the library.

Working with Libraries or modules can be one of the more difficult realms of computer programming. Libraries are pre written blocks of computer and come in a wide variety of sizes and formats from small single file libraries to complete development frameworks.

Each individual library can come with its own style, function naming and methods that may differ from the underlying native language. Becoming familiar with and learning a library can take time. The fact that there are so many libraries available, each with its strengths and weaknesses and each library designed for specific tasks can make it difficult to choose a library for your project. There is no easy way around this other than reading up on the library use and examples, looking at forums and asking others about their experience with the library, or just vetting a few that closely match the solution you require and testing them yourself.

Implementing or "including" a library into your project is not overly difficult, but can come with unforeseen quirks, especially when implementing more than one library in your project. Libraries are often built on previously created libraries and naming conflicts can occur occasionally. Not all library methods are compatible with other library methods so as an example mixing a windows GUI (windowing) library with direct calls to the OS API window creation can run into conflicts.

Another issue with libraries is the platform in which they are intended to be used. Some libraries are "Cross-Platform" and will compile on both Windows and Unix systems, while others may only be used on an individual platform. Most significant libraries are cross-platform, for example GTK2+, wxWidgets, SDL2.

Static vs Runtime libraries.

Static libraries are included (or merged) into the application source code before the application is transferred to RAM and executed. In compiled portable executables that are assembled into the final .exe, the static library is included into the final .exe. Static libraries will always be allocated a share of RAM as part of the running **.exe**. When multiple applications use the same library, each application creates its own duplicate of the library in its runtime space in RAM.

Runtime libraries, or dynamically linked libraries (DLL), are compiled into a .dll, or in the case of a Linux system, a "Shared Object" **.so**. Runtime library modules, DLLs, are called from our application only when needed during the application's execution, and are released from RAM when no longer

needed. Many applications can call the same DLL at runtime, removing the need to transport the entire library with each application. The OS manages how each application gets a share of the DLLs resources and time allocation. Each method has pros and cons, and comes down to the programmer's choice to meet the requirements of the final application.

Order of module includes

It is important to pay attention to the order in which we include libraries. When a library or module is included, the entire source of the library/s exists at the top of your source code. In basic terms everything is written as a single source file at compile time or run time. If the library files are correctly written then it should not matter what order you include them in the head of your source but as a general rule include system headers (includes) first followed by standard library headers, then 3rd party library headers followed by your project header files.

Some will say to use the reverse order to what I have described and this is true if you are writing actual Library files and want to ensure they are self-contained. If the order of included headers creates an error in a library file then it is highly likely that the library is poorly created.

Due to the size of the subject I will cover the use of libraries as well as Inter-process Communication (IPC) in the more language specific book titles. Inter-process Communication is the term given to interacting with other applications. Many run-time libraries are executable files and require commands sent via pipes or streams. The command line makes use of 3 pipes; stdin, stdout and stderr and is a common path for IPC. Applications such as SQLite and TCL/tk are examples where we interact with another application via IPC.

Cross platform coding.

Cross platform coding means that your code will compile or run on different OS platforms. This requires writing the code lines needed for any platform in which you require your code to work. To achieve this we use *if-else* decisions throughout our code to direct the program flow to the correct lines of code for the OS in use.

```
If OS == UNIX Then
    REM Place UNIX code here
Else If OS == Windows Then
    REM Place Windows code here
End If
```

The std library is standard across Windows and Unix systems, but is very limited. It does not take long until we need to make use of OS dependent requests from the Operating System. In this case a different function call will be required depending upon the OS that the source is used on at the time.

Application design and flow charts.

Application design is somewhat synonymous with programming. This is the conceptualisation and design stage of programming. First we have a problem or an idea, and then we go about implementing a solution. Start broad and progressively refine your solutions into blocks of smaller solutions.

You may start by just scribbling out a basic concept of what you want to achieve without adding any detail.

For example I want to build a calculator.

Next we will think about what components we may need; for example:

- A GUI with buttons and a numerical display panel.
- Mathematical functions for our calculations.
- Somewhere to store our data in the calculator.

Next break down the above into smaller blocks.

GUI:

- Use TCL/TK (Tinkter) library for the GUI
- Main window at 200W px by 130H px.
- 20 buttons at 30Wpx x 25 px.
- Text area 160W x 50H

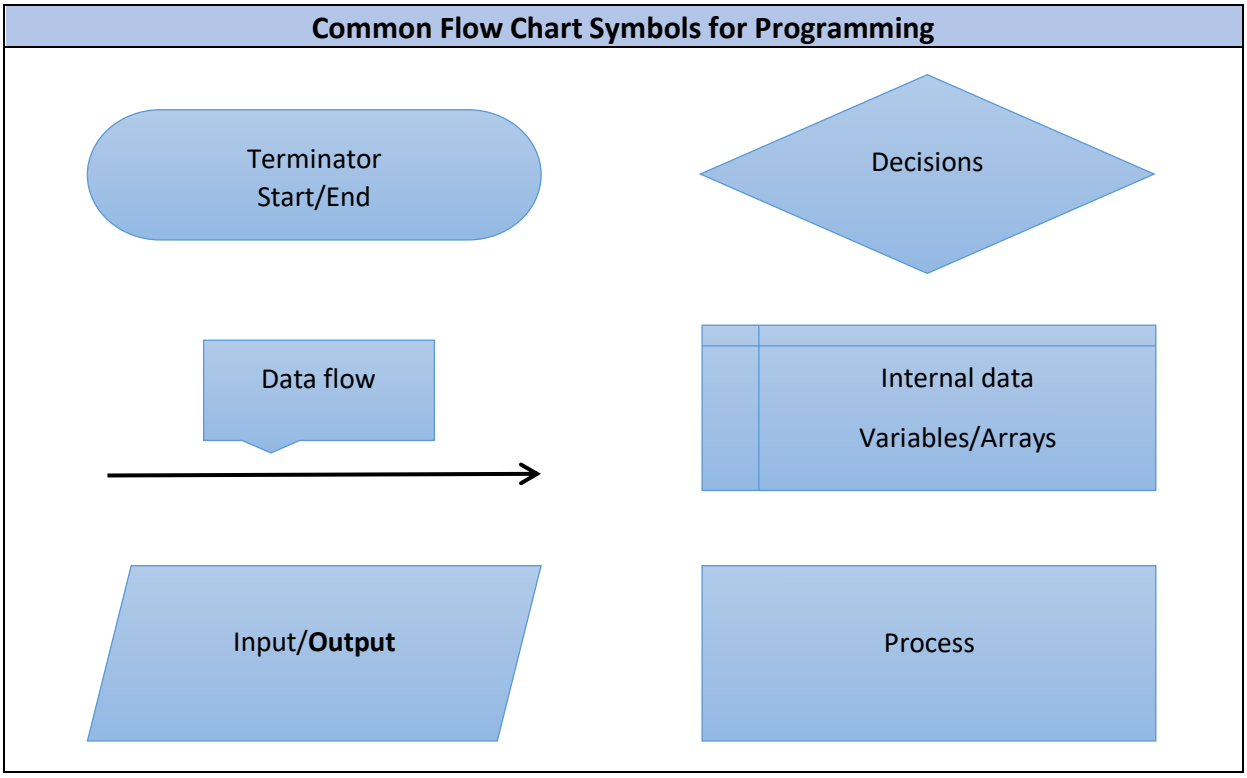
From here we can begin to create the flow and logic of our application.

- When Button 1 is pressed send numeral one to text area.
- If Arithmetic Operator is pressed store content of text area followed by Operator.
- If numeric button pressed send numeral to text area.
- If Arithmetic Operator is pressed evaluate content of text area with previous stored value and store the result.
- If = pressed evaluate content of text area with previous stored value and display the result in the text area.

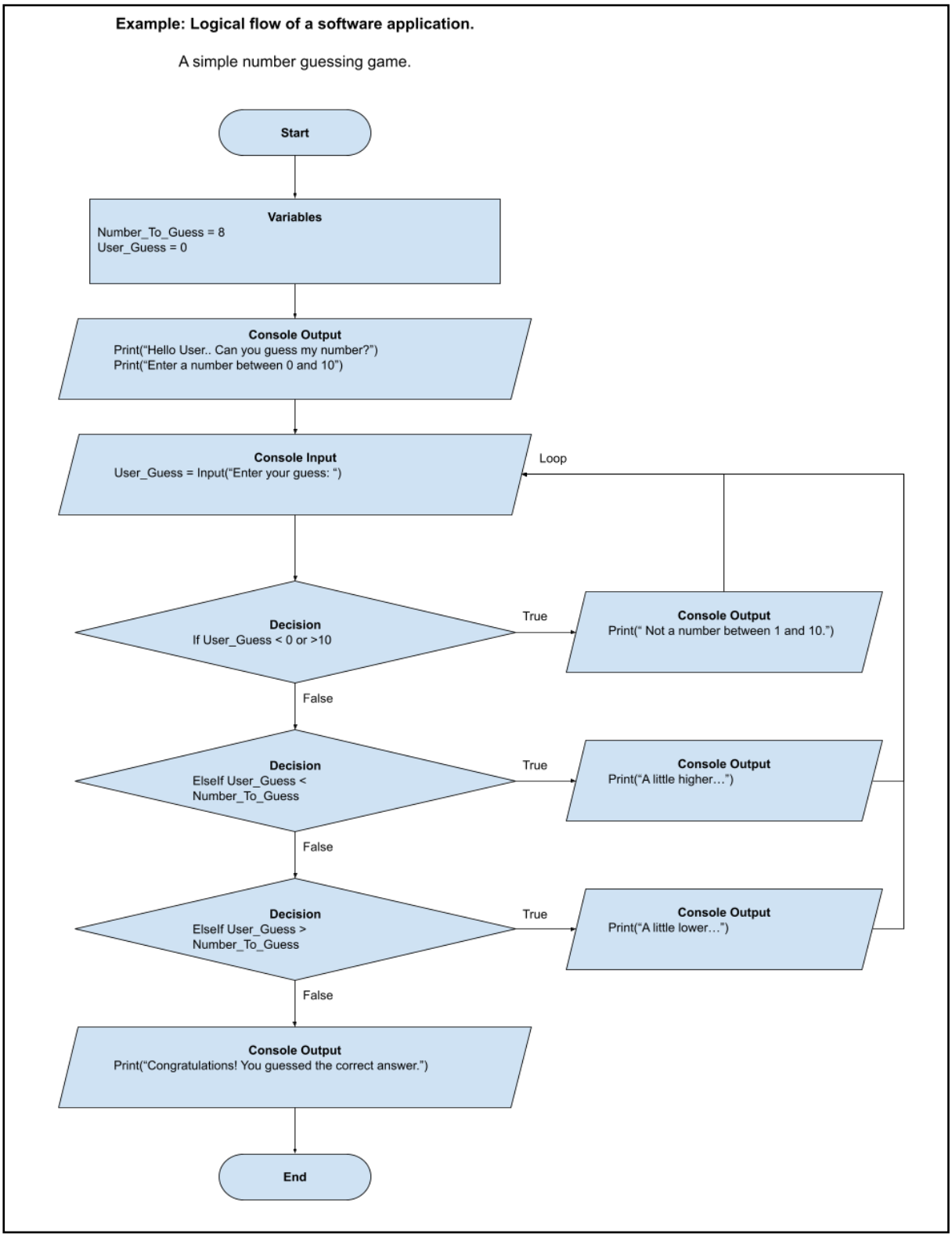
Flowcharts play an important part in the development design cycle but can also be distracting and time consuming when we are first learning to code. Don't be too consumed with creating flow charts while learning as we really need to understand the program flow and logic first which only comes from jumping in and coding and attempting different logic. Code your heart out and the principles of logic and flow will come in time. That being said when you do come to the point of creating your own significant application be sure at a minimum to write down the steps above and set out a basic outline of the problem and solution you may wish to implement. I sometimes even draw mind maps with pictures on a few pieces of paper first just to create an "Overall Picture" of the project before jumping into any formal outline.

Flow charts, logic and application design are generally independent of the platform or coding language that is used in the final product. As I have shown in all of the examples we can implement a solution in most common languages and across multiple platforms based upon our original required outcomes.

Flow diagrams:



The following is the flowchart that lays out the programmatic flow and the logic of the simple guessing game from an earlier example. Flow charts can become large and complex very quickly and it is common to break down individual blocks of the application into separate flow charts.



An example of a flow chart used earlier in the document.

Modularity

Modularity is the technique of creating blocks of code, often as functions, that can be reused within the same application as well as other applications without any modification.

Throughout the examples I have included a number of “Helper Functions” which are commonly called throughout the main body of the code. Instead of writing 6 to 10 lines of code every time I need to clear the screen, I can just use a single line call to Con_Clear(). All of the function calls used from the standard library and most other libraries are an example of modular code.

Debugging

Debugging is the art of finding and correcting errors within our code and code that does not produce the outcome that we expect. There are a number of tools and methods for finding errors and tracing the expected output of our application. It is important to make every effort to create error free applications as well as applications that produce the results that are required in a consistent way.

It is close to impossible to write code that works as expected and without errors the first time we run it. Coders will often spend as much time debugging code as they had writing the code in the first place so this is not unusual.

Application bugs can occur from a large number of directions including sources outside of the actual application. Predicting all of the possibilities is an almost impossible task. The most common bugs’ will come from our own code and will include 2 different categories. Bugs that are directly related to syntax and coding errors that result in the application failing to run or crashing during execution, and bugs where the application and code runs fine except we get results that are outside of what we expected.

Most software languages will ship with a built in debugger and syntax check, and bugs that are related to syntax and coding errors will most often be caught by the debugger. Errors and warnings will be displayed in a debug console with the type of error, the line in which it appeared to have occurred and often the column that appears close to the problem. In these cases it is often just a syntax error where we have misspelt a keyword or mistypes the code punctuation. Read the debug output warnings carefully as they will often show an error number and a brief description. You can search the error number and description in google and will often get hints from the language forums or from the language documentation itself.

The second type of bug is related to unexpected results from our software. Finding bugs of this nature requires us to methodically step through our code monitoring the values of our variables and arrays as the application is running. There are 3 common methods for achieving this; using blocking or non blocking print statements, using the built in break points of the language and using unit tests.

Text formatting is also a common bug issue. It is easy to forget to include or exclude new line characters ‘\n’, escape characters “C:\\mydir\\myfile.txt” or format the print output of an integer as a string. If you are getting unexpected characters or a print statement gives a blank like take a close look at the print format for the text it is to display.

Debug Prints

Using print statements is a common method of debugging. This is most commonly used when our application runs without showing debug error warnings but doesn’t execute as expected or returns unexpected results.

We add a print statement before as well as a print statement after the position in the application that contains the error and progressively move the print statements toward each other in the body of the application. This may mean starting with one print at the very first line of the application and a print statement as the very last line if we don't have any clues as to where the error is occurring.

If the application is stalling (Stops running part way through) we can use a simple "Non Blocking Print" **`Print("DEBUG – Before"); Print("DEBUG – After")`**. This will allow the program to run until the error occurs and we will see the fist debug print but not the second. Move the first print down in steps until it no longer displays when the application is run, then move it back a few lines until the first debug displays again. Move the second debug print up to a few lines after the point where the fist debug print stopped displaying.

You have now narrowed down a few lines or a block of code where the error is occurring. Read through the code between each print and look for obvious code errors and take note of what variables and the data that is being manipulated. You can continue to narrow the print statements closer in until you have just 1 or 2 lines, but this may not be workable inside of a loop.

If the print statement sits above and below a function call, pay close attention to the arguments being sent to the function. Here we can use a "Blocking Print" to halt the application when it enters the function and visually check the values of the variables being sent as arguments.

```
Print("DEBUG my_varable = %d", my_variable);
System("pause");
```

Test each variable value and ensure that they display what is expected. If they don't, trace the variable back up through the application with the "Blocking Print" checking the return value of the variable after any expression or manipulation of the variable's value until you find the problem expression.

If all of the variables values being sent to the function call as arguments are correct, you may need to "step into" the function and apply the same techniques to check the function for errors. If it is a large application and a complex function it may be less time consuming to test the function on its own in a separate source code document.

Debugging inside of loops can be tedious and difficult. It's just the way it is. Most common bugs within loops are due to what are known as "bounds errors'' and can often lead to a "Segmentation Fault" or segfault. Bound errors occur when we attempt to enumerate data elements of an array that are outside of the boundary of the actual number of elements allocated to the array. Keep in mind text data as strings are essentially just arrays of characters.

If the array has 12 elements and we attempt to access the 13^{th} element we will encounter an error as it does not exist. Sometimes this error will occur silently creating what appears like an endless loop with a blank screen, or can even read data from the memory of another application or segfault if there is an attempt to write data to another application.

Use a blocking debug print inside of the loop and "step" through each repeat of the loop displaying the value of any loop counters to see if they go out of bounds. Take note of the boundaries. An array-like data structure is often created with *n* elements and accessing the elements will be *n-1*.

The following will access the correct number of elements from 0 to 4 in a 5 element array.

```
my_array[5]
```

```
For x = 0 To 4 Step 1
    Print my_array[x]
Next x
```

The following will access too many elements from 0 to 5 in a 5 element array, causing an error.

```
my_array[5]
Length = Len(my_array)  REM Will return 5
For x = 0 To Length Step 1
    Print my_array[x]  REM my_array[5] is out of bounds.
Next x
```

To correct this we need to correct the bounds alignment.

```
my_array[5]
Length = Len(my_array)  REM Will return 5
For x = 0 To Length -1 Step 1
    Print my_array[x]  REM my_array[4] is now the last element.
Next x
```

Enumerating an array from 1 To 5 will also cause an alignment issue.

```
For x = 1 To 5 Step 1
```

Some programming languages will allow us to create a data structure of elements with 5 elements and enumerate the elements from 1 To 5, while others will create a data structure of elements with 5 elements and enumerate the elements from 0 To 4, so it is important to take note of the how the data element is constructed and enumerated for that particular language.

We can also include a debug print inside of a loop to monitor how the actual data is being manipulated to see if it is as expected.

The following will make an exact copy of the data from the first array to the second array.

```
first_array[5]
second_array[5]
len_arrays = 5 -1
For x = 0 To len_arrays Step 1
    second_array[x] = first_array[x]
Next x
```

If at some point later in our application second_array is not displaying the data as we had expected, we can go back and test if it is being copied correctly.

```
first_array[5]
second_array[5]
len_arrays = 5 -1
For x = 0 To len_arrays Step 1
    second_array[x] = first_array[x]
    Print “DEBUG: f_array = “, first _array [x], “s_array = “,
second_array[x]
    System(“pause”)
Next x
```

We can now step through each enumeration and compare what data has been copied.

Debug prints are one of the most common debugging techniques and can be used almost anywhere. Another type of debug print is the “Pop-Up GUI print dialog”. This uses the same technique except a GUI window displays our debug information instead of printing it on a console line. This can be useful when we are trying to correct console print format issues and don’t want additional print statements messing up the output, or working in a TUI or GUI environment that has no console output. I have included a number of GUI debug prints in the examples in this booklet.

Debugging is a skill that takes time to learn and be proficient at. Be patient as it is a skill that develops over time as you learn to program.

Examples

In the following I am attempting to populate a 2 dimensiona array using the variable’s integer values during the loop. You will see that the final values are off by 1 and range from 5 to 9 instead of 6 to 10 except for the FreeBASIC example which counts 6 values. In this example the problem is reasonable obvious in that the count range is from 0 to 4 and adding the **Col_x +5** does not give the values that we require. I will step through the array values using a Print statement as an example of what it may appear like in a more complex application.

Language: C

Code

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

// Print a table 5 rows x 5 columns
// with each column numbered 6 to 10
int main(void)
    {
    int ROWS = 5;
    int COLUMNS = 5;
    int Row_y = 0;
    int Col_x = 0;
    static int MyArray[5][5];

    for (Row_y = 0; Row_y < ROWS; Row_y++)
        {
        for (Col_x = 0; Col_x < COLUMNS; Col_x++)
            {
            MyArray[Row_y][Col_x] = Col_x +5;
            }
        }

    for (Row_y = 0; Row_y < ROWS; Row_y++)
        {
        for (Col_x = 0; Col_x < COLUMNS; Col_x++)
            {
            printf("%d",MyArray[Row_y][Col_x]);
            }
            printf("\n");
```

```
        }

return 0;
    }
```

Output: Counts from 5 to 9

```
56789
56789
56789
56789
56789

--------------------------------
Process exited after 0.0278 seconds with return value 0
Press any key to continue . . .
```

Let's add a Print debug to watch the variable values being entered. I have added a second debug print to assist with format and readability.

Code:

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

// Print a table 5 rows x 5 columns
// with each column numbered 6 to 10
int main(void)
    {
    int ROWS = 5;
    int COLUMNS = 5;
    int Row_y = 0;
    int Col_x = 0;
    static int MyArray[5][5];

    for (Row_y = 0; Row_y < ROWS; Row_y++)
        {
        for (Col_x = 0; Col_x < COLUMNS; Col_x++)
            {
            printf("DEBUG Col_x = %d ", Col_x);// DEBUG
            MyArray[Row_y][Col_x] = Col_x +5;
            }
            printf("\n");// DEBUG Newline
        }

    for (Row_y = 0; Row_y < ROWS; Row_y++)
        {
        for (Col_x = 0; Col_x < COLUMNS; Col_x++)
            {
            printf("%d",MyArray[Row_y][Col_x]);
            }
            printf("\n");
        }

return 0;
    }
```

Output: Variable enumerates 0 to 4

```
DEBUG_Col_x = 0 DEBUG_Col_x = 1 DEBUG_Col_x = 2 DEBUG_Col_x = 3 DEBUG_Col_x = 4
DEBUG_Col_x = 0 DEBUG_Col_x = 1 DEBUG_Col_x = 2 DEBUG_Col_x = 3 DEBUG_Col_x = 4
DEBUG_Col_x = 0 DEBUG_Col_x = 1 DEBUG_Col_x = 2 DEBUG_Col_x = 3 DEBUG_Col_x = 4
DEBUG_Col_x = 0 DEBUG_Col_x = 1 DEBUG_Col_x = 2 DEBUG_Col_x = 3 DEBUG_Col_x = 4
DEBUG_Col_x = 0 DEBUG_Col_x = 1 DEBUG_Col_x = 2 DEBUG_Col_x = 3 DEBUG_Col_x = 4
56789
56789
56789
56789
56789

--------------------------------
Process exited after 0.03615 seconds with return value 0
Press any key to continue . . .
```

Now let’s change our print statement to show what values are entered into each array element.

Code:

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

// Print a table 5 rows x 5 columns
// with each column numbered 6 to 10
int main(void)
    {
    int ROWS = 5;
    int COLUMNS = 5;
    int Row_y = 0;
    int Col_x = 0;
    static int MyArray[5][5];

    for (Row_y = 0; Row_y < ROWS; Row_y++)
        {
        for (Col_x = 0; Col_x < COLUMNS; Col_x++)
            {
            printf("DEBUG_Col_x = %d ", Col_x +5);// DEBUG
            MyArray[Row_y][Col_x] = Col_x +5;
            }
            printf("\n");// DEBUG Newline
        }

    for (Row_y = 0; Row_y < ROWS; Row_y++)
        {
        for (Col_x = 0; Col_x < COLUMNS; Col_x++)
            {
            printf("%d",MyArray[Row_y][Col_x]);
            }
            printf("\n");
        }

return 0;
    }
```

Output:

```
DEBUG_Col_x = 5 DEBUG_Col_x = 6 DEBUG_Col_x = 7 DEBUG_Col_x = 8 DEBUG_Col_x = 9
DEBUG_Col_x = 5 DEBUG_Col_x = 6 DEBUG_Col_x = 7 DEBUG_Col_x = 8 DEBUG_Col_x = 9
DEBUG_Col_x = 5 DEBUG_Col_x = 6 DEBUG_Col_x = 7 DEBUG_Col_x = 8 DEBUG_Col_x = 9
DEBUG_Col_x = 5 DEBUG_Col_x = 6 DEBUG_Col_x = 7 DEBUG_Col_x = 8 DEBUG_Col_x = 9
DEBUG_Col_x = 5 DEBUG_Col_x = 6 DEBUG_Col_x = 7 DEBUG_Col_x = 8 DEBUG_Col_x = 9
56789
56789
56789
56789
56789

--------------------------------
Process exited after 0.03044 seconds with return value 0
Press any key to continue . . .
```

We can easily see that our for loop is enumerated from 0 to 4 and adding 5 to Col_x only brings us to 6 to 9 for the array entries. So the array counts 5 positions 0 to4 and we want the real numbers 1 to 5 and will have to alter the value of Col_x +1 to correct the alignment.

In the next example I have added an extra +1 to correct the array count to real numbers. I have also moved the Debug Print statement to reflect the values after they have been entered into the array to confirm that the correct value is entered into each element.

Code:

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

// Print a table 5 rows x 5 columns
// with each column numbered 6 to 10
int main(void)
    {
    int ROWS = 5;
    int COLUMNS = 5;
    int Row_y = 0;
    int Col_x = 0;
    static int MyArray[5][5];

    for (Row_y = 0; Row_y < ROWS; Row_y++)
        {
        for (Col_x = 0; Col_x < COLUMNS; Col_x++)
            {
            MyArray[Row_y][Col_x] = Col_x +1 +5;
            printf("DEBUG_MyArray = %d ", MyArray[Row_y][Col_x]);// DEBUG
            }
            printf("\n");// DEBUG Newline
        }

    for (Row_y = 0; Row_y < ROWS; Row_y++)
        {
        for (Col_x = 0; Col_x < COLUMNS; Col_x++)
            {
            printf("%d",MyArray[Row_y][Col_x]);
            }
            printf("\n");
        }

return 0;
```

```
    }
```

Output:

```
DEBUG_MyArray = 6 DEBUG_MyArray = 7 DEBUG_MyArray = 8 DEBUG_MyArray = 9 DEBUG_MyArray = 10
DEBUG_MyArray = 6 DEBUG_MyArray = 7 DEBUG_MyArray = 8 DEBUG_MyArray = 9 DEBUG_MyArray = 10
DEBUG_MyArray = 6 DEBUG_MyArray = 7 DEBUG_MyArray = 8 DEBUG_MyArray = 9 DEBUG_MyArray = 10
DEBUG_MyArray = 6 DEBUG_MyArray = 7 DEBUG_MyArray = 8 DEBUG_MyArray = 9 DEBUG_MyArray = 10
DEBUG_MyArray = 6 DEBUG_MyArray = 7 DEBUG_MyArray = 8 DEBUG_MyArray = 9 DEBUG_MyArray = 10
678910
678910
678910
678910
678910

--------------------------------
Process exited after 0.03456 seconds with return value 0
Press any key to continue . . .
```

We can see that the correction has worked and now enters the numbers from 6 to 10.

We can now delete the debug print statements as we no longer have use for them. If it is a complex source code and you find that you have multiple Print debugs you can just comment them out in case you need to come back and recheck values at another stage in the application creation. I have labelled my debug prints with the word “DEBUG” in every instance as this makes it easy to do a keyword search and remove any debug lines before exporting the completed code.

Normally best practice would be to create 2 additional variables to avoid magic numbers:

```
int array_align = 1;
int shift_five = 5;
```

Code:

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

// Print a table 5 rows x 5 columns
// with each column numbered 6 to 10
int main(void)
    {
    int ROWS = 5;
    int COLUMNS = 5;
    int Row_y = 0;
    int Col_x = 0;
    int array_align = 1;  // Easier to understand than magic numbers :)
    int shift_five = 5;  // Easier to understand than magic numbers :)

    static int MyArray[5][5];

    for (Row_y = 0; Row_y < ROWS; Row_y++)
        {
        for (Col_x = 0; Col_x < COLUMNS; Col_x++)
            {
            MyArray[Row_y][Col_x] = Col_x +array_align +shift_five;
            }
        }

    for (Row_y = 0; Row_y < ROWS; Row_y++)
        {
        for (Col_x = 0; Col_x < COLUMNS; Col_x++)
            {
```

```
            printf("%d",MyArray[Row_y][Col_x]);
            }
            printf("\n");
        }

return 0;
    }
```

Output:

```
678910
678910
678910
678910
678910

--------------------------------
Process exited after 0.02698 seconds with return value 0
Press any key to continue . . .
```

Language: FreeBASIC

Code

```
Declare Function main_procedure() As Integer
main_procedure()
' Print a table 5 rows x 5 columns
' with each column numbered 6 to 10
Function main_procedure() As Integer
    Dim As UInteger ROWS = 5
    Dim As UInteger COLUMNS = 5
    Dim As UInteger Row_y = 0
    Dim As UInteger Col_x = 0
    Dim As UInteger MyArray(5, 5)

    For Row_y = 0 To ROWS Step 1    ' Count through each row.
        For Col_x = 0 to COLUMNS Step 1
            MyArray(Row_y, Col_x) = Col_x +5
        Next Col_x
    Next Row_y

    for Row_y = 0 to ROWS Step 1    ' Count through each row.
        for Col_x = 0 To COLUMNS Step 1
            Print MyArray(Row_y, Col_x);
        Next Col_x
        Print ""
    Next Row_y

    Print "Press any key to continue..."
    Sleep  ' Sleep until a key is pressed
    Return 0
End Function  'main_procedure
```

Output: Counts from 5 to 10

```
5678910
5678910
5678910
5678910
5678910
5678910
Press any key to continue...
```

Let's add a Print debug to watch the variable values being entered. I have added a second debug print to assist with format and readability.

Code:

```
Declare Function main_procedure() As Integer
main_procedure()
' Print a table 5 rows x 5 columns
' with each column numbered 6 to 10
Function main_procedure() As Integer
    Dim As UInteger ROWS = 5
    Dim As UInteger COLUMNS = 5
    Dim As UInteger Row_y = 0
    Dim As UInteger Col_x = 0
    Dim As UInteger MyArray(5, 5)

    For Row_y = 0 To ROWS Step 1    ' Count through each row.
        For Col_x = 0 to COLUMNS Step 1
            Print "DEBUG_Col_x = ";Col_x;" ";   ' DEBUG
            MyArray(Row_y, Col_x) = Col_x +5
        Next Col_x
        Print ""   ' DEBUG Newline
    Next Row_y

    for Row_y = 0 to ROWS Step 1    ' Count through each row.
        for Col_x = 0 To COLUMNS Step 1
            Print MyArray(Row_y, Col_x);
        Next Col_x
        Print ""
    Next Row_y

    Print "Press any key to continue..."
    Sleep  ' Sleep until a key is pressed
    Return 0
End Function  'main_procedure
```

Output:

```
DEBUG_Col_x = 0 DEBUG_Col_x = 1 DEBUG_Col_x = 2 DEBUG_Col_x = 3 DEBUG_Col_x = 4 DEBUG_Col_x = 5
DEBUG_Col_x = 0 DEBUG_Col_x = 1 DEBUG_Col_x = 2 DEBUG_Col_x = 3 DEBUG_Col_x = 4 DEBUG_Col_x = 5
DEBUG_Col_x = 0 DEBUG_Col_x = 1 DEBUG_Col_x = 2 DEBUG_Col_x = 3 DEBUG_Col_x = 4 DEBUG_Col_x = 5
DEBUG_Col_x = 0 DEBUG_Col_x = 1 DEBUG_Col_x = 2 DEBUG_Col_x = 3 DEBUG_Col_x = 4 DEBUG_Col_x = 5
DEBUG_Col_x = 0 DEBUG_Col_x = 1 DEBUG_Col_x = 2 DEBUG_Col_x = 3 DEBUG_Col_x = 4 DEBUG_Col_x = 5
DEBUG_Col_x = 0 DEBUG_Col_x = 1 DEBUG_Col_x = 2 DEBUG_Col_x = 3 DEBUG_Col_x = 4 DEBUG_Col_x = 5
5678910
5678910
5678910
5678910
5678910
5678910
Press any key to continue...
```

I can quickly see that I have 2 problems here:
1. I am enumerating 6 array elements when I have only declared 5 (Potential "Out of bounds/Overflow error")

2. As with the previous example I will need to offset the array enumeration with a +1 to create real numbers.

I will correct the enumeration by reducing the for loop steps end value COLUMNS -1 as well as add the +1 offset to create real numbers from our for next loop counters.

Code:

```
Declare Function main_procedure() As Integer
main_procedure()
' Print a table 5 rows x 5 columns
' with each column numbered 6 to 10
Function main_procedure() As Integer
    Dim As UInteger ROWS = 5
    Dim As UInteger COLUMNS = 5
    Dim As UInteger Row_y = 0
    Dim As UInteger Col_x = 0
    Dim As UInteger MyArray(5, 5)

    For Row_y = 0 To ROWS -1 Step 1    ' Count through each row.
        For Col_x = 0 to COLUMNS -1 Step 1
            Print "DEBUG_Col_x = ";Col_x;" ";   ' DEBUG
            MyArray(Row_y, Col_x) = Col_x +1 +5
        Next Col_x
        Print ""   ' DEBUG Newline
    Next Row_y

    for Row_y = 0 to ROWS -1 Step 1    ' Count through each row.
        for Col_x = 0 To COLUMNS -1 Step 1
            Print MyArray(Row_y, Col_x);
        Next Col_x
        Print ""
    Next Row_y

    Print "Press any key to continue..."
    Sleep  ' Sleep until a key is pressed
    Return 0
End Function  'main_procedure
```

Output:

```
DEBUG_Col_x = 0 DEBUG_Col_x = 1 DEBUG_Col_x = 2 DEBUG_Col_x = 3 DEBUG_Col_x = 4
DEBUG_Col_x = 0 DEBUG_Col_x = 1 DEBUG_Col_x = 2 DEBUG_Col_x = 3 DEBUG_Col_x = 4
DEBUG_Col_x = 0 DEBUG_Col_x = 1 DEBUG_Col_x = 2 DEBUG_Col_x = 3 DEBUG_Col_x = 4
DEBUG_Col_x = 0 DEBUG_Col_x = 1 DEBUG_Col_x = 2 DEBUG_Col_x = 3 DEBUG_Col_x = 4
DEBUG_Col_x = 0 DEBUG_Col_x = 1 DEBUG_Col_x = 2 DEBUG_Col_x = 3 DEBUG_Col_x = 4
678910
678910
678910
678910
678910
Press any key to continue...
```

We can see that the correction has worked and now enters the numbers from 6 to 10.

I am not going to test the values that are entered into the array as I did with the C example above as the output clearly shows that the array is outputting the correct values.

We can now delete the debug print statements as we no longer have use for them. If it is a complex source code and you find that you have multiple Print debugs you can just comment them out in case you need to come back and recheck values at another stage in the application creation. I have labelled my debug prints with the word “DEBUG” in every instance as this makes it easy to do a keyword search and remove any debug lines before exporting the completed code.

Normally best practice would be to create 2 additional variables to avoid magic numbers:

```
Dim As UInteger array_align = 1
Dim As UInteger shift_five = 5
```

Code:

```
Declare Function main_procedure() As Integer
main_procedure()
' Print a table 5 rows x 5 columns
' with each column numbered 6 to 10
Function main_procedure() As Integer
    Dim As UInteger ROWS = 5
    Dim As UInteger COLUMNS = 5
    Dim As UInteger Row_y = 0
    Dim As UInteger Col_x = 0
    Dim As UInteger MyArray(5, 5)
    Dim As UInteger array_align = 1
    Dim As UInteger shift_five = 5

    For Row_y = 0 To ROWS -array_align Step 1    ' Count through each row.
        For Col_x = 0 to COLUMNS -array_align Step 1
            MyArray(Row_y, Col_x) = Col_x +array_align +shift_five
        Next Col_x
    Next Row_y

    for Row_y = 0 to ROWS -array_align Step 1    ' Count through each row.
        for Col_x = 0 To COLUMNS -array_align Step 1
            Print MyArray(Row_y, Col_x);
        Next Col_x
        Print ""
    Next Row_y

    Print "Press any key to continue..."
    Sleep  ' Sleep until a key is pressed
    Return 0
End Function  'main_procedure
```

Output:

```
678910
678910
678910
678910
678910
Press any key to continue...
```

Language: Python3

Code

```
# Print a table 5 rows x 5 columns
# with each column numbered 6 to 10
def main():
    ROWS = 5
    COLUMNS = 5
    Row_y = 0
    Col_x = 0
    MyArray = [[None]* COLUMNS for _ in range(ROWS)]

    for Row_y in range(0,  ROWS):
        for Col_x in range(0,  COLUMNS):
            MyArray[Row_y][Col_x] = Col_x +5

    for Row_y in range(0,  ROWS):
        for Col_x in range(0,  COLUMNS):
            print(MyArray[Row_y][Col_x], end="")
        print("")

    input("Press [Enter] to exit.")
    return None

if __name__ == '__main__':
    main()
```

Output: Counts from 5 to 9

```
56789
56789
56789
56789
56789
Press [Enter] to exit.
```

Let's add a Print debug to watch the variable values being entered. I have added a second debug print to assist with format and readability.

Code:

```
# Print a table 5 rows x 5 columns
# with each column numbered 6 to 10
def main():
    ROWS = 5
    COLUMNS = 5
    Row_y = 0
    Col_x = 0
    MyArray = [[None]* COLUMNS for _ in range(ROWS)]

    for Row_y in range(0,  ROWS):
        for Col_x in range(0,  COLUMNS):
            print("DEBUG Col_x = ", Col_x, " ", end="")  # DEBUG
            MyArray[Row_y][Col_x] = Col_x +5
        print("")  # DEBUG Newline

    for Row_y in range(0,  ROWS):
        for Col_x in range(0,  COLUMNS):
            print(MyArray[Row_y][Col_x], end="")
        print("")
```

```
    input("Press [Enter] to exit.")
    return None

if __name__ == '__main__':
    main()
```

Output:

```
DEBUG_Col_x =  0  DEBUG_Col_x =  1  DEBUG_Col_x =  2  DEBUG_Col_x =  3  DEBUG_Col_x =  4
DEBUG_Col_x =  0  DEBUG_Col_x =  1  DEBUG_Col_x =  2  DEBUG_Col_x =  3  DEBUG_Col_x =  4
DEBUG_Col_x =  0  DEBUG_Col_x =  1  DEBUG_Col_x =  2  DEBUG_Col_x =  3  DEBUG_Col_x =  4
DEBUG_Col_x =  0  DEBUG_Col_x =  1  DEBUG_Col_x =  2  DEBUG_Col_x =  3  DEBUG_Col_x =  4
DEBUG_Col_x =  0  DEBUG_Col_x =  1  DEBUG_Col_x =  2  DEBUG_Col_x =  3  DEBUG_Col_x =  4
56789
56789
56789
56789
56789
Press [Enter] to exit.
```

We can easily see that our for loop is enumerated from 0 to 4 and adding 5 to Col_x only brings us to 6 to 9 for the array entries. So the array counts 5 positions 0 to 4 and we want the real numbers 1 to 5 and will have to alter the value of Col_x +1 to correct the alignment.

Code:

```
# Print a table 5 rows x 5 columns
# with each column numbered 6 to 10
def main():
    ROWS = 5
    COLUMNS = 5
    Row_y = 0
    Col_x = 0
    MyArray = [[None]* COLUMNS for _ in range(ROWS)]

    for Row_y in range(0,  ROWS):
        for Col_x in range(0,  COLUMNS):
            print("DEBUG_Col_x = ", Col_x +1, " ", end="")  # DEBUG
            MyArray[Row_y][Col_x] = Col_x +1 +5
        print("")  # DEBUG Newline

    for Row_y in range(0,  ROWS):
        for Col_x in range(0,  COLUMNS):
            print(MyArray[Row_y][Col_x], end="")
        print("")

    input("Press [Enter] to exit.")
    return None

if __name__ == '__main__':
    main()
```

Output:

```
DEBUG_Col_x =  1  DEBUG_Col_x =  2  DEBUG_Col_x =  3  DEBUG_Col_x =  4  DEBUG_Col_x =  5
DEBUG_Col_x =  1  DEBUG_Col_x =  2  DEBUG_Col_x =  3  DEBUG_Col_x =  4  DEBUG_Col_x =  5
DEBUG_Col_x =  1  DEBUG_Col_x =  2  DEBUG_Col_x =  3  DEBUG_Col_x =  4  DEBUG_Col_x =  5
DEBUG_Col_x =  1  DEBUG_Col_x =  2  DEBUG_Col_x =  3  DEBUG_Col_x =  4  DEBUG_Col_x =  5
DEBUG_Col_x =  1  DEBUG_Col_x =  2  DEBUG_Col_x =  3  DEBUG_Col_x =  4  DEBUG_Col_x =  5
678910
678910
678910
678910
678910
Press [Enter] to exit._
```

We can now delete the debug print statements as we no longer have use for them. If it is a complex source code and you find that you have multiple Print debugs you can just comment them out in case you need to come back and recheck values at another stage in the application creation. I have labelled my debug prints with the word "DEBUG" in every instance as this makes it easy to do a keyword search and remove any debug lines before exporting the completed code.

Normally best practice would be to create 2 additional variables to avoid magic numbers:

```
array_align = 1
shift_five = 5
```

Code:

```
# Print a table 5 rows x 5 columns
# with each column numbered 6 to 10
def main():
    ROWS = 5
    COLUMNS = 5
    Row_y = 0
    Col_x = 0
    array_align = 1
    shift_five = 5
    MyArray = [[None]* COLUMNS for _ in range(ROWS)]

    for Row_y in range(0,  ROWS):
        for Col_x in range(0,  COLUMNS):
            MyArray[Row_y][Col_x] = Col_x +array_align +shift_five

    for Row_y in range(0,  ROWS):
        for Col_x in range(0,  COLUMNS):
            print(MyArray[Row_y][Col_x], end="")
        print("")

    input("Press [Enter] to exit.")
    return None

if __name__ == '__main__':
    main()
```

Output:

```
678910
678910
678910
678910
678910
Press [Enter] to exit._
```

So far I have only used “Non-Blocking” Debug Print statements. Non-blocking means that the application will continue to run without stopping or pausing at any of the Debug Print outputs.

The second common method is known as a “Blocking Debug Print”. A blocking Debug print will halt the execution of the application at the position of the Blocking statement until the user issues a response to allow the application to continue. This is most often achieved by placing an “Input” or a “Pause” statement directly following the “Debug Print”. This is particularly useful in large applications allowing us to execute our code up until just after where we think the problem line of code exists. It can also allow us to identify where a problem line in our source code is by moving the blocking print up or down in our source. A second form of “Blocking Debug” is the use of a GUI Dialog box often referred to as a MessageBox. A message box will send the values you choose to a pop up window which will require user interaction to continue; normally by pressing the [OK] button.

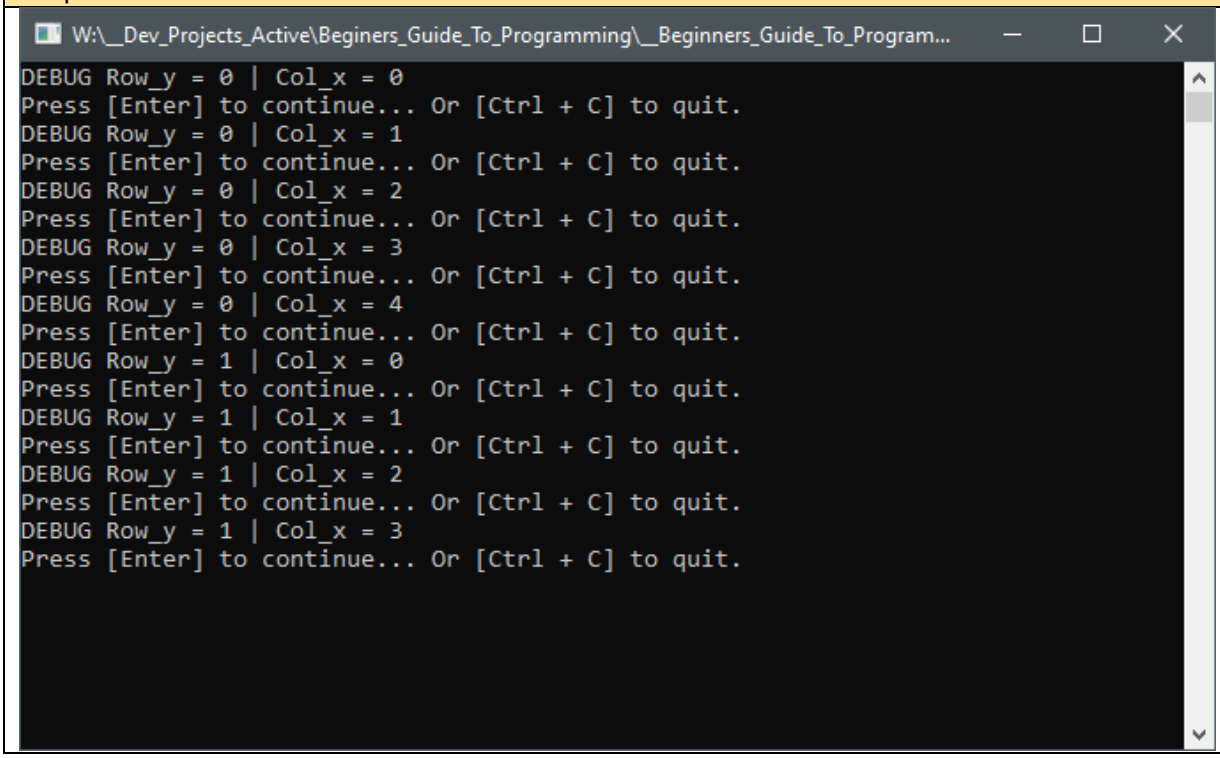
The following examples will show the use of “Blocking Debug Print” statements. I have created some “Safe” and simple “Wrapper” functions to “Pause” the execution until [Enter] is pressed. Any form of input or system(“pause”) line is effective.

Code: C Blocking

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

int Dbg_Pause(void);
// Print a table 5 rows x 5 columns
// with each column numbered 6 to 10
int main(void)
    {
    int ROWS = 5;
    int COLUMNS = 5;
    int Row_y = 0;
    int Col_x = 0;

    static int MyArray[5][5];

    for (Row_y = 0; Row_y < ROWS; Row_y++)
        {
        for (Col_x = 0; Col_x < COLUMNS; Col_x++)
            {
            MyArray[Row_y][Col_x] = Col_x +5;
            printf("DEBUG Row_y = %d | Col_x = %d \n", Row_y, Col_x);  // DEBUG
Print
            Dbg_Pause();  // DEBUG Pause (Blocking)
            }
        }

    for (Row_y = 0; Row_y < ROWS; Row_y++)
        {
        for (Col_x = 0; Col_x < COLUMNS; Col_x++)
            {
            printf("%d",MyArray[Row_y][Col_x]);
            }
            printf("\n");
        }

return 0;
    }
```

```c
// Safe getchar() removes all artifacts from the stdin buffer.
// Modified version of S_Pause() and S_getchar().
int Dbg_Pause(void)  // DEBUG
    {
    printf("Press [Enter] to continue... Or [Ctrl + C] to quit.");
    // This function is referred to as a wrapper for getchar()
    int i = 0;
    int ret;
    int ch;
    // The following enumerates all characters in the buffer.
    while((ch = getchar()) != '\n' && ch != EOF )
        {
        // But only keeps and returns the first char.
        if (i < 1)
            {
            ret = ch;
            }
        i++;
        }
    return ret;
    }
```

Output:

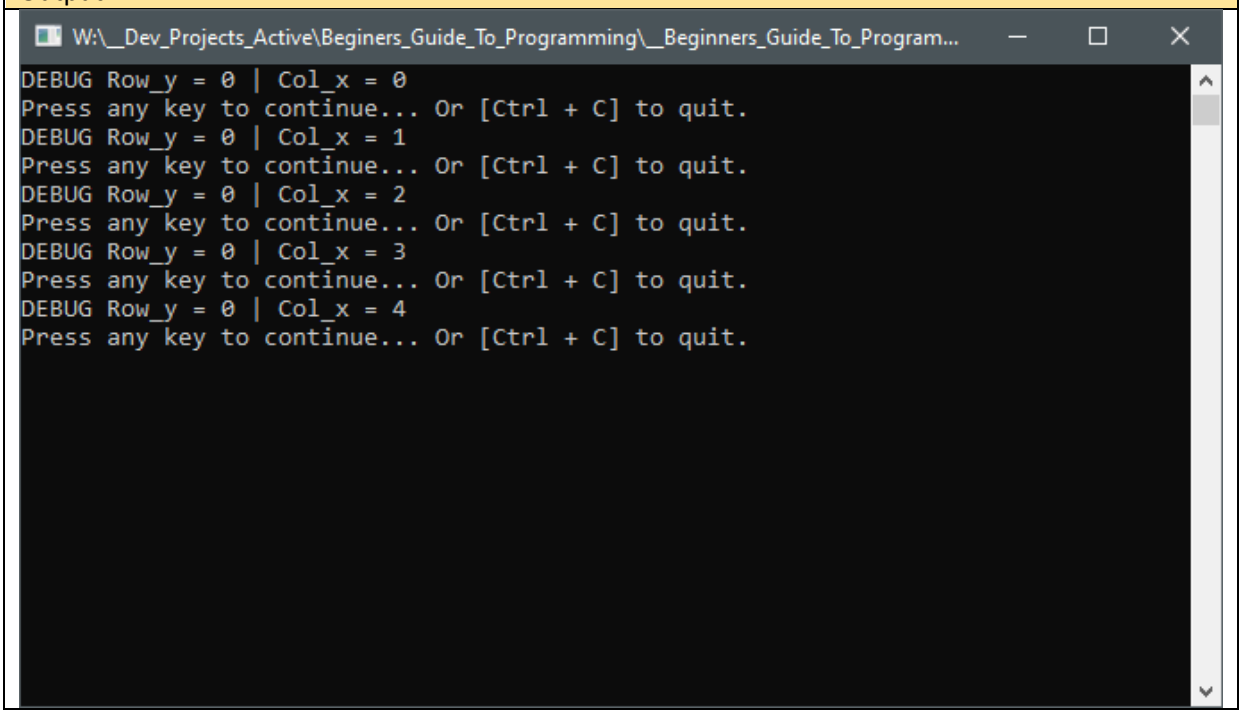


Code: FreeBASIC Blocking

```
Declare Function main_procedure() As Integer
Declare Function Dbg_Pause() As Integer

main_procedure()
' Print a table 5 rows x 5 columns
' with each column numbered 6 to 10
Function main_procedure() As Integer
```

```
    Dim As UInteger ROWS = 5
    Dim As UInteger COLUMNS = 5
    Dim As UInteger Row_y = 0
    Dim As UInteger Col_x = 0
    Dim As UInteger MyArray(5, 5)

    For Row_y = 0 To ROWS -1 Step 1    ' Count through each row.
        For Col_x = 0 to COLUMNS -1 Step 1
            MyArray(Row_y, Col_x) = Col_x +5
            Print "DEBUG Row_y = ";Row_y; " | Col_x = "; Col_x   ' DEBUG Print
            Dbg_Pause()  ' DEBUG Pause (Blocking)
        Next Col_x
    Next Row_y

    for Row_y = 0 to ROWS -1 Step 1    ' Count through each row.
        for Col_x = 0 To COLUMNS -1 Step 1
            Print MyArray(Row_y, Col_x);
        Next Col_x
        Print ""
    Next Row_y

    Print "Press any key to continue..."
    Sleep  ' Sleep until a key is pressed
    Return 0
End Function  'main_procedure

' Console Pause (GetKey version) DEBUG
Function Dbg_Pause() As Integer
    Dim As Long dummy
    Print !"Press any key to continue... Or [Ctrl + C] to quit."
    dummy = Getkey
    Return 0
End Function
```

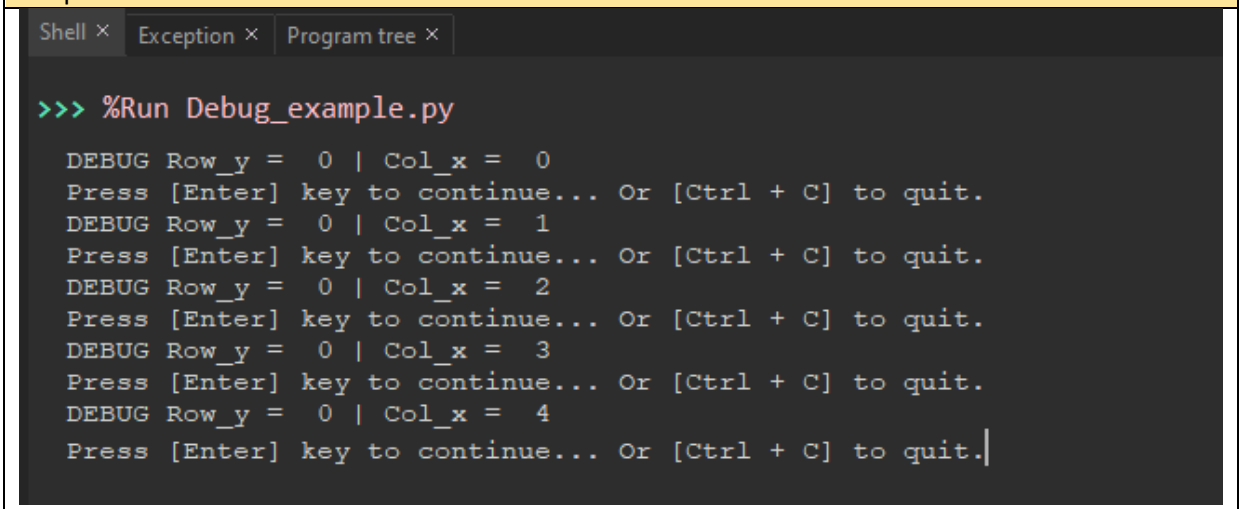
Output:

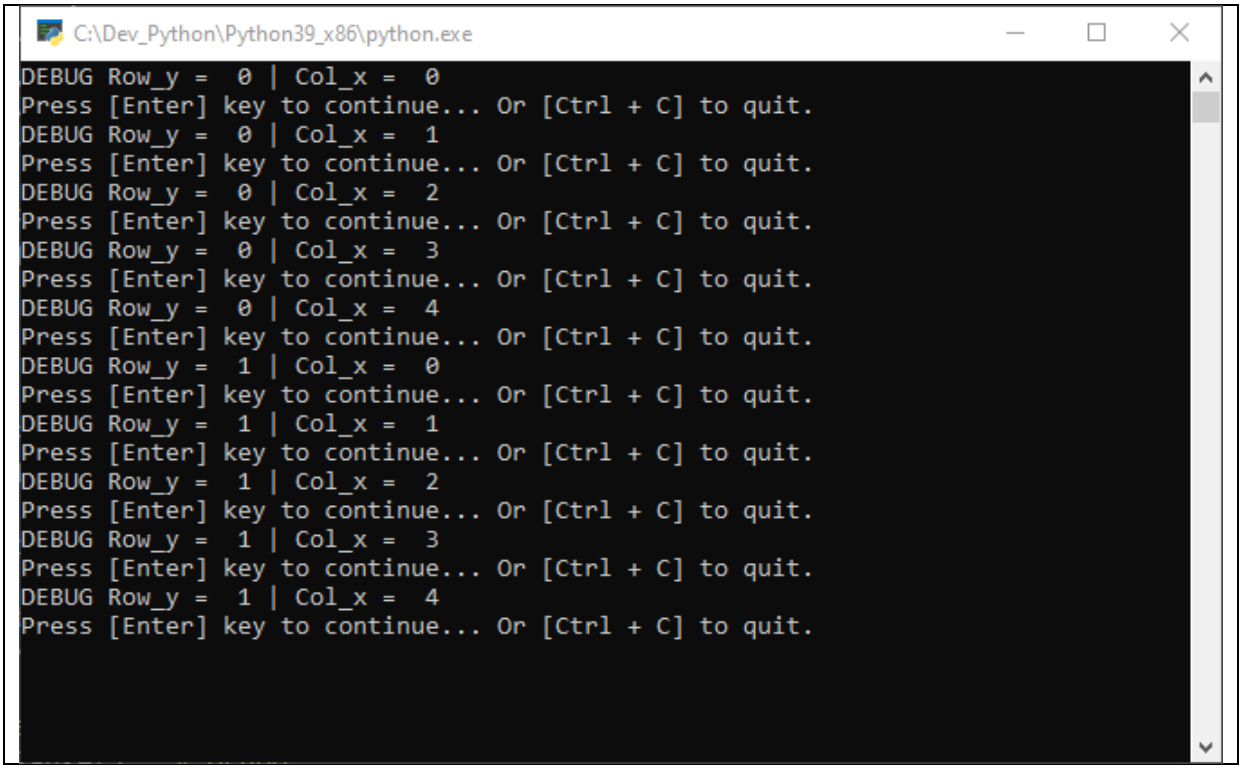


Code: Python 3 Blocking

```
# Print a table 5 rows x 5 columns
# with each column numbered 6 to 10
def main():
    ROWS = 5
    COLUMNS = 5
    Row_y = 0
    Col_x = 0
    MyArray = [[None]* COLUMNS for _ in range(ROWS)]

    for Row_y in range(0,  ROWS):
        for Col_x in range(0,  COLUMNS):
            MyArray[Row_y][Col_x] = Col_x +5
            print("DEBUG Row_y = ", Row_y, "| Col_x = ", Col_x)  # DEBUG Print
            Dbg_Pause()  # DEBUG Pause (Blocking)

    for Row_y in range(0,  ROWS):
        for Col_x in range(0,  COLUMNS):
            print(MyArray[Row_y][Col_x], end="")
        print("")

    input("Press [Enter] to exit.")
    return None

# Console Pause wrapper.
def Dbg_Pause():  # DEBUG
    dummy = ""
    dummy = input("Press [Enter] key to continue... Or [Ctrl + C] to quit.")
    return None

if __name__ == '__main__':
    main()
```

Output:





The GUI dialog is specific to the OS library and APIs so I need to include the extra libraries and a Function “Wrapper” for convenience. The following example contains a cross platform DebugMsg() helper function to reduce the amount of code required to add a DEBUG Message output in our code.

I often have many different helper functions that I keep in a collection to assist while coding. Although a more advanced topic I normally keep them in a library file and include the file at the top of my source #include “Helper_Functions.c”. It is just one line in the header of the source and is easy to delete when finished. Alternatively you can copy paste your debug and helper functions to the bottom of your source code and delete them when you are happy the code is working as expected.

The DebugMsg() function looks larger and more bloated than it actually is due to the extra comments to help you understand how the GUI calls and error returns are working.

Laguage: C – CLI/GUI
Code example: “Debug_example.c”

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

// Platform specific headers.
// Test if Windows or Linux OS
#ifdef _WIN32
#include <Windows.h>
#define OS_Windows 1  // 1 = True (aka Bool)
#define OS_Unix 0
#elif __linux__ // __unix__
#include <unistd.h>
#define OS_Unix 1
#define OS_Windows 0  // 0 = False (aka Bool)
#else
#error "OS Not Supported!"
#include <stophere>
```

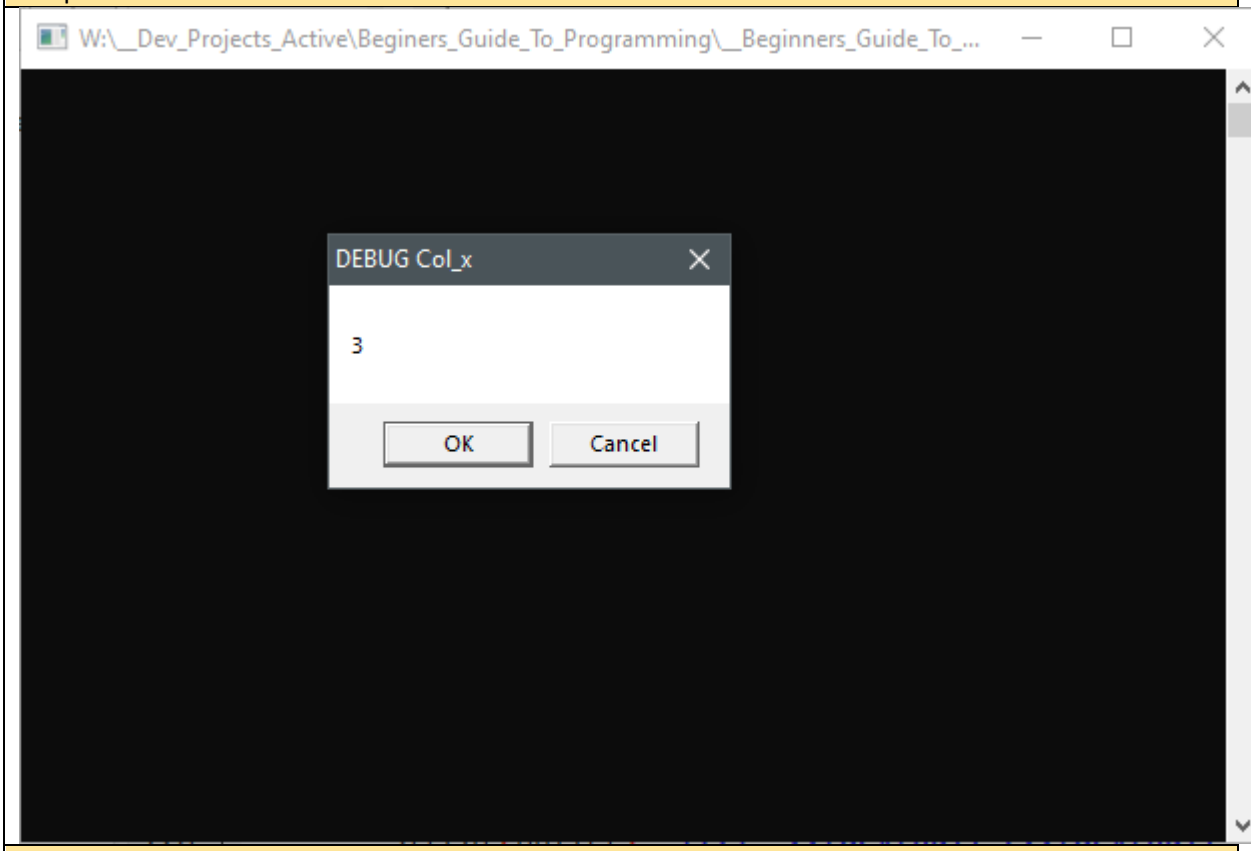
```
#endif

int DebugMsg(char *aTitle, char *aMessage);
// Print a table 5 rows x 5 columns
// with each column numbered 6 to 10
int main(void)
    {
    int ROWS = 5;
    int COLUMNS = 5;
    int Row_y = 0;
    int Col_x = 0;
    static int MyArray[5][5];

    for (Row_y = 0; Row_y < ROWS; Row_y++)
        {
        for (Col_x = 0; Col_x < COLUMNS; Col_x++)
            {
            MyArray[Row_y][Col_x] = Col_x +5;
            // DEBUG Lines START -->
            // C Is a low level language so we have to know the type and convert
            // it to a string. In this case we are converting an int to a
            // char array aka String. "%d" is for int, "%f" is for float.
            char str_buffer[128] = {'\0'};  // Max string/num length 128 chars.
            sprintf(str_buffer, "%d", Col_x);  // Convert int Number to String.
            DebugMsg("DEBUG Col_x", str_buffer);  // send converted number.
            // DEBUG Lines END <--
            }
        }

    for (Row_y = 0; Row_y < ROWS; Row_y++)
        {
        for (Col_x = 0; Col_x < COLUMNS; Col_x++)
            {
            printf("%d",MyArray[Row_y][Col_x]);
            }
        printf("\n");
        }
    return 0;
    }


// A simple message box for debugging.
// Takes a String value only, so you will have to convert numbers to String.
int DebugMsg(char *aTitle, char *aMessage)
    {
    // Require a temporary buffer to hold converted string.
    // Number conversions. Convert Int, Float, bin, hex to ASCII String.
    //
    // int sprintf(char *str, const char *format, ...)
    // NOTE! sprintf uses the same print formatting as printf...
    // Search "C library function - sprintf()" for the full description.
    //
    // int iVariable = 238;
    //char Buffer[128] = {'\0'};
    //sprintf(Buffer, "%d", iVariable) // "%f" float|double float.
    //DebugMsg("DEBUGmsg", Buffer);
    int reterr = 0;  // Holds the return value from the command line.

    if(OS_Windows)
```

```
        {
        // Requires:winuser.h (#include <Windows.h>),User32.dll
        // Not attached to parent console window.
        // https://docs.microsoft.com/en-us/windows/win32/api/winuser
        // /nf-winuser-messageboxa
        // May throw a compiler warning... MessageBoxA Not found.
        reterr = MessageBoxA(0, aMessage, aTitle, 1);  // 65536, MB_SETFOREGROUND
        if(reterr == 0)  // Holds the return value from the command line.
            {
            // MessageBox() Fail
            return 0;
            }
        else if(reterr == 1)
            {
            // IDOK
            return 1;
            }
        else if(reterr == 2)
            {
            // IDCANCEL (Ctrl + C is dissabled when GUI messagebox is used)
            // So provide an option to break out of the debugging...
            // This will also exit now if the Close X is selected.
            exit(0);  // Clean exit the application.
            }
        else
            {
            return -1;
            }
        // To attach to the parent console window.
        //MessageBoxA(FindWindowA("ConsoleWindowClass", NULL),msg,title,0);
        }
    else if(OS_Unix)
        {
        // http://manpages.ubuntu.com/manpages/trusty/man1/xmessage.1.html
        // apt-get install x11-utils
        //system("xmessage -center 'Hello, World!'");
        // Else try wayland
        // https://github.com/Tarnyko/wlmessage
        //system("wlmessage 'Hello, World!'");
        char Buffer[128] = {'\0'};
        char Buf_Msg[128] = {'\0'};
        // Place title text in 'apostrophe'.
        strcpy(Buf_Msg, "\'");
        strcat(Buf_Msg, aTitle);
        strcat(Buf_Msg, "\'");

        // Build our command line statement.
        // xmessage [-options] [message ...]
        strcpy(Buffer, "xmessage -buttons OK:101,CANCEL:102 -center -title ");
        strcat(Buffer, Buf_Msg);
        strcat(Buffer, " \'.:|" );
        strcat(Buffer, aMessage );
        // NOTE! ">>/dev/null 2>>/dev/null" suppresses the console output.
        strcat(Buffer, "|:.\' >>/dev/null 2>>/dev/null" );
        // Send it to the command line.
        reterr = system(Buffer);
        /*
                // This is the recommended linux way of getting the return from
calling
```

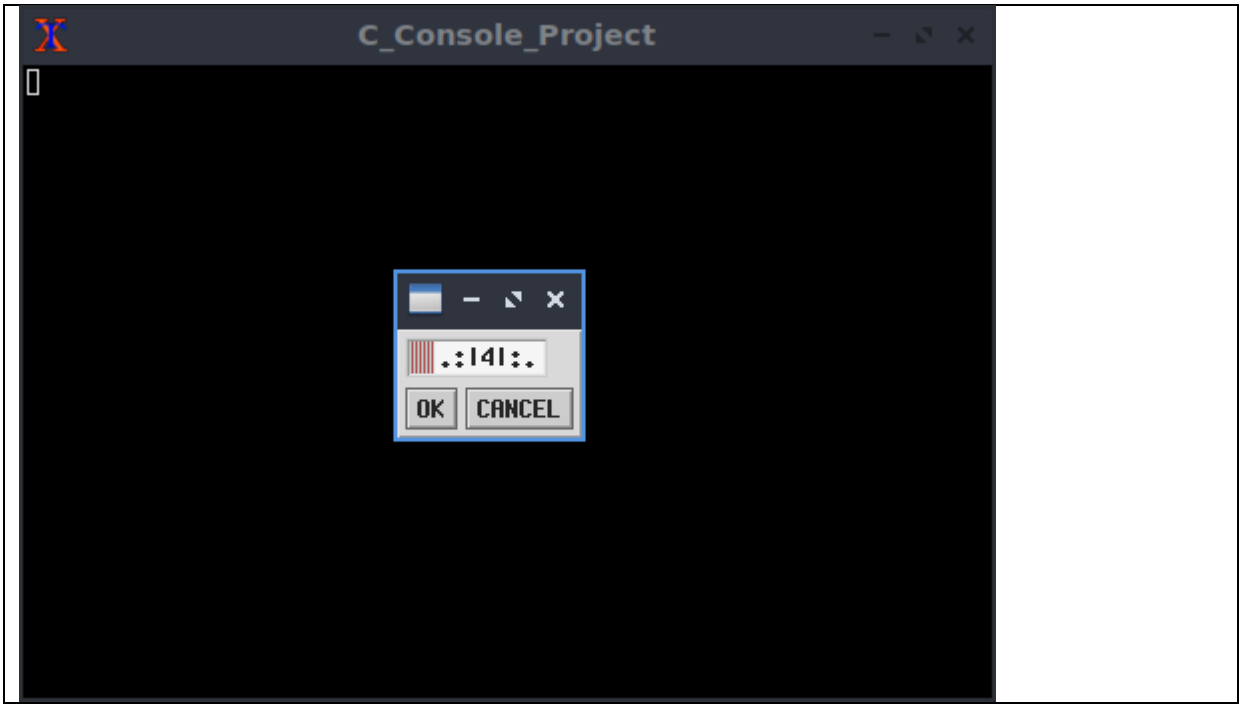
```
                // a command line application. It has the overhead of including
wait.h
                // under the Linux OS test...
                //#elif __linux__
                //#include <wait.h>

                if ( WIFEXITED(reterr))
                {
                printf("The return value: %d\n", WEXITSTATUS(reterr));
                }
                else if (WIFSIGNALED(reterr))
                {
                printf("The program exited because of signal (signal no:%d)\n",
WTERMSIG(reterr));
                }
        */
        // As the return value is stored in the "High Byte" of the 16bit error
return,
        // we can use some bitwise shift magic to extract the value of the 8 bit
high
        // byte and eliminate the need for the wait.h library file.
        if(reterr>>8 == 101)
            {
            // ID OK
            //printf("reterr 101 = %d\n", reterr>>8);
            }
        else if(reterr>>8 == 102)
            {
            // ID CANCEL
            //printf("reterr 102 = %d\n", reterr>>8);
            exit(0);  // Clean exit the application.
            }
        else if(reterr>>8 == 1)
            {
            // ID CLOSE X
            //printf("reterr 1 = %d\n", reterr>>8);
            }
        else if(reterr>>8 == 0)
            {
            // ID Default run OK
            //printf("reterr Default = %d\n", reterr>>8);
            }
        else
            {
            // FAIL! Try Wayland...
            // I have left the Wayland version as a default message as
            // few systems are using wayland at this time. To add the cancel
            // routines just follow the -buttons OK:010,CANCEL:102 and error
            // tests as per xmessage.
            // Try Wayland compositor wlmessage.
            strcpy(Buffer, "wlmessage \' |" );
            strcat(Buffer, aMessage );
            strcat(Buffer, "| \' >>/dev/null 2>>/dev/null" );
            reterr = system(Buffer);
            if((reterr != 0) && (reterr != 1))
                {
                // Popup message failed.
                //printf("%d\n", reterr>>8);
                return -1;
```

```
                }
            }
        // All above If tests will default to this return 0; upon success.
        return 0;
        }
    else
    {
        // OS Unknown
    }

    return 0;
    }
```

Output: Windows



Output: Linux



Laguage: FreeBASIC – CLI/GUI

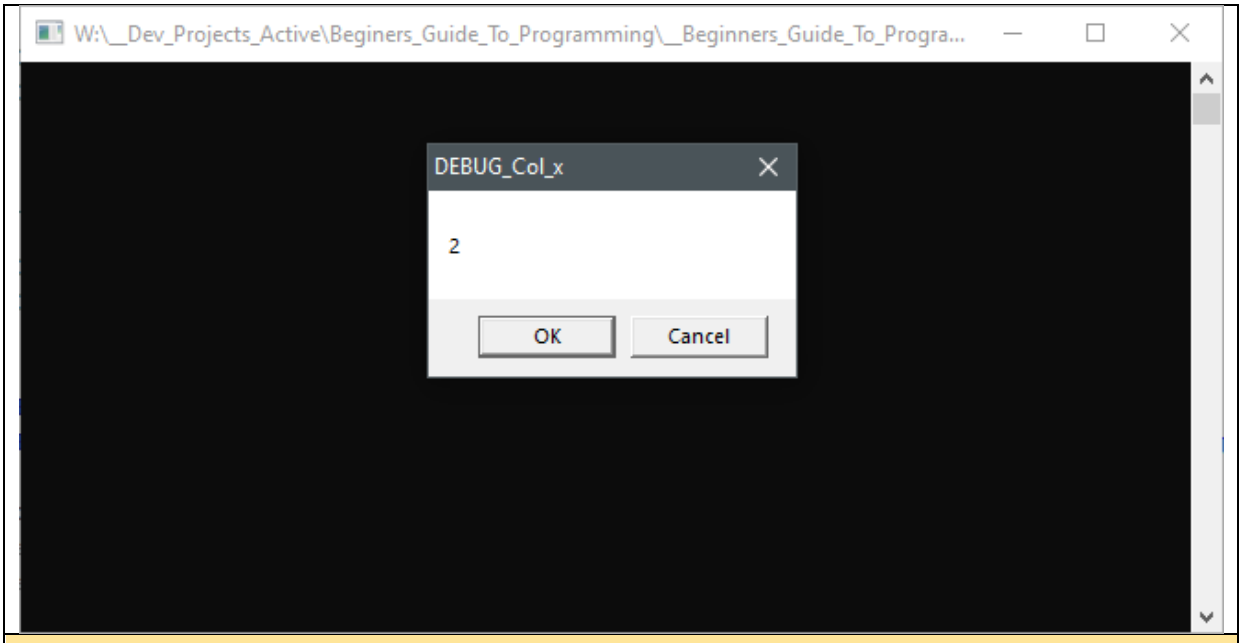
Code example: “Debug_example.bas”

```
' Test if Windows or Unix OS
#ifdef __FB_WIN32__
#include once "windows.bi"
#define OS_Windows 1  ' 1 = True (aka Bool)
#define OS_Unix 0
#endif

#ifdef __FB_UNIX__'__FB_LINUX__
' TODO
#define OS_Unix 1
#define OS_Windows 0  ' 0 = False (aka Bool)
#endif

Declare Function main_procedure() As Integer
Declare Function DebugMsg(Byref aTitle As String, Byref aMessage As String) As
Integer

main_procedure()
' Print a table 5 rows x 5 columns
' with each column numbered 6 to 10
Function main_procedure() As Integer
    Dim As UInteger ROWS = 5
    Dim As UInteger COLUMNS = 5
    Dim As UInteger Row_y = 0
    Dim As UInteger Col_x = 0
    Dim As UInteger MyArray(5, 5)

    For Row_y = 0 To ROWS -1 Step 1    ' Count through each row.
        For Col_x = 0 to COLUMNS -1 Step 1
            MyArray(Row_y, Col_x) = Col_x +5
            ' Unlike C BASIC is a high level language and we can simply
            ' use Str() to convert a number to a string.
```

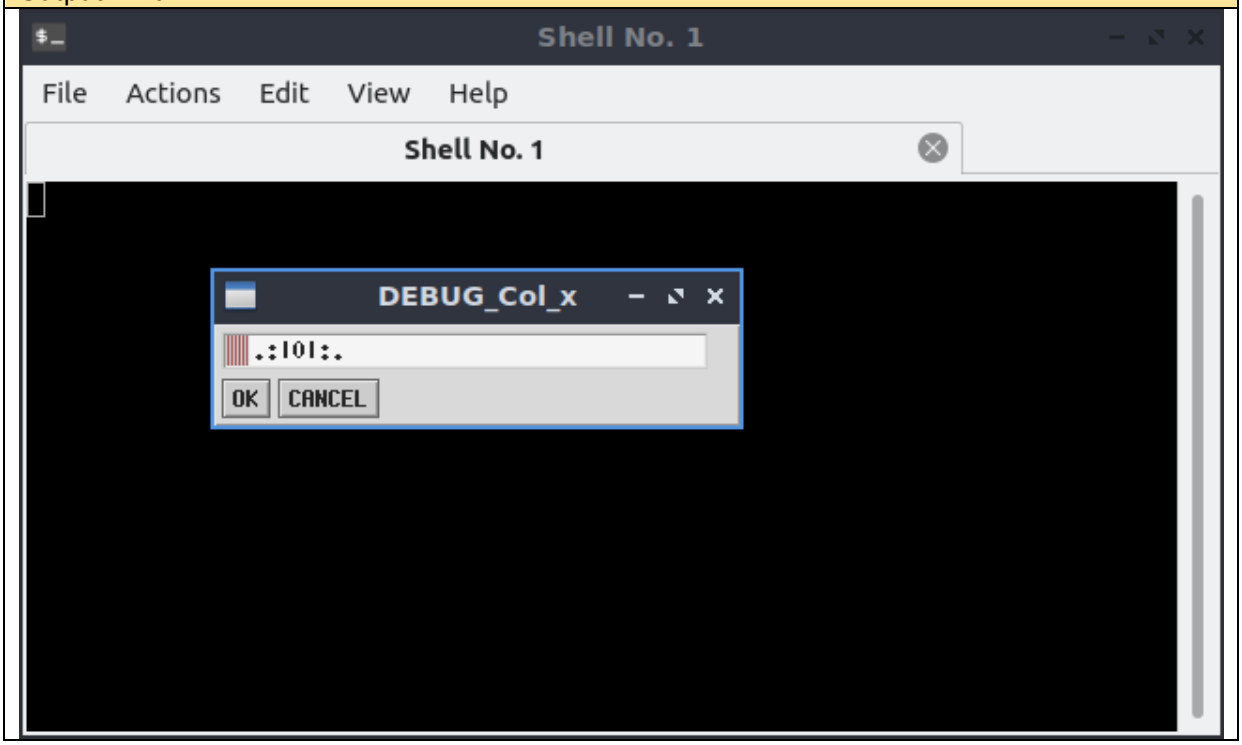
```
            DebugMsg("DEBUG_Col_x", Str(Col_x))
        Next Col_x
    Next Row_y

    for Row_y = 0 to ROWS -1 Step 1    ' Count through each row.
        for Col_x = 0 To COLUMNS -1 Step 1
            Print MyArray(Row_y, Col_x);
        Next Col_x
        Print ""
    Next Row_y

    Print "Press any key to continue..."
    Sleep  ' Sleep until a key is pressed
    Return 0
End Function  'main_procedure

' A simple message box for debugging.
' Takes a String value only, so you will have to convert numbers to String.
Function DebugMsg(Byref aTitle As String, Byref aMessage As String) As Integer
    ' Requires:"windows.bi" (#include once "windows.bi").
    ' Not attached to parent console window.
    ' https://docs.microsoft.com/en-us/windows/win32/api/winuser
    ' /nf-winuser-messageboxa
    ' May throw a compiler warning... MessageBoxA Not found.

    If(OS_Windows = 1) Then
        #ifdef __FB_WIN32__
        Dim As Integer reterr = 0  ' Holds the return value from the command line.
        reterr = MessageBox(NULL, aMessage, aTitle, MB_OKCANCEL)
        If reterr = 0 Then
            ' MessageBox() Fail
            Return 0
        Elseif reterr = 1 then
            ' IDOK
            Return 1
        Elseif reterr = 2 then
            ' IDCANCEL (Ctrl +C is dissabled when GUI messagebox is used)
            ' So provide an option to break out of the debugging...
            ' This will also exit now if the Close X is selected.
            ' Please note that the "End" statement may not clean up all memory
            ' and its use is discouraged. Only use it in debugging and
            ' not in production code.
            End
        else
            Return -1
        End If
        ' To attached to parent console window.
        'MessageBoxA(FindWindowA("ConsoleWindowClass", NULL),msg,title,0);
        #endif
    Elseif(OS_Unix = 1) Then
        ' http://manpages.ubuntu.com/manpages/trusty/man1/xmessage.1.html
        ' apt-get install x11-utils
        'system("xmessage -center 'Hello, World!'");
        ' Else try wayland
        ' https://github.com/Tarnyko/wlmessage
        'system("wlmessage 'Hello, World!'");

            Dim As Integer reterr = 0  ' Holds the return value from the command
line.
```

```
        Dim As String Buffer = ""
        Dim As String Buf_Msg = ""
        ' Place title text in 'apostrophe'.
        Buf_Msg = "'" & aTitle & "'"

        ' Build our command line statement.
        ' xmessage [-options] [message ...]
        ' NOTE! ">>/dev/null 2>>/dev/null" suppresses the console output.
        Buffer = "xmessage -buttons OK:101,CANCEL:102 -center -title " & Buf_Msg &
" '.:|" & aMessage & "|:.' >>/dev/null 2>>/dev/null"
             ' Sometimes the application gets to the message window before the
console
             ' window has fully opened. This leaves the message as a background
window.
             ' Commonly referred to as a "Race Condition", so I have placed a
1/10
             ' second sleep to give the parent console window time to open.
             Sleep 100
        ' Send it to the command line.
        reterr = Shell(Buffer)
        If(reterr = 101) Then
            ' ID OK
        ElseIf(reterr = 102) Then
            ' ID CANCEL
            End  ' Quit debugging
        ElseIf(reterr = 1) Then
            ' ID CLOSE X
        ElseIf(reterr = 0) Then
            ' ID Default run OK
        Else  ' xmessage failed or not exist.
            ' Try Wayland compositor wlmessage.
            Buffer = "wlmessage ' |" & aMessage & "| ' >>/dev/null 2>>/dev/null"
            reterr = Shell(Buffer)
            If(reterr <> 0) And (reterr <> 1) Then
                ' Popup message failed.
                'Print  reterr
                Return -1
            End If
        End If
    Else
        ' OS Unknown
        Return -1
    End If
    Return 0
End Function
```

Output: Windows



Output: Linux



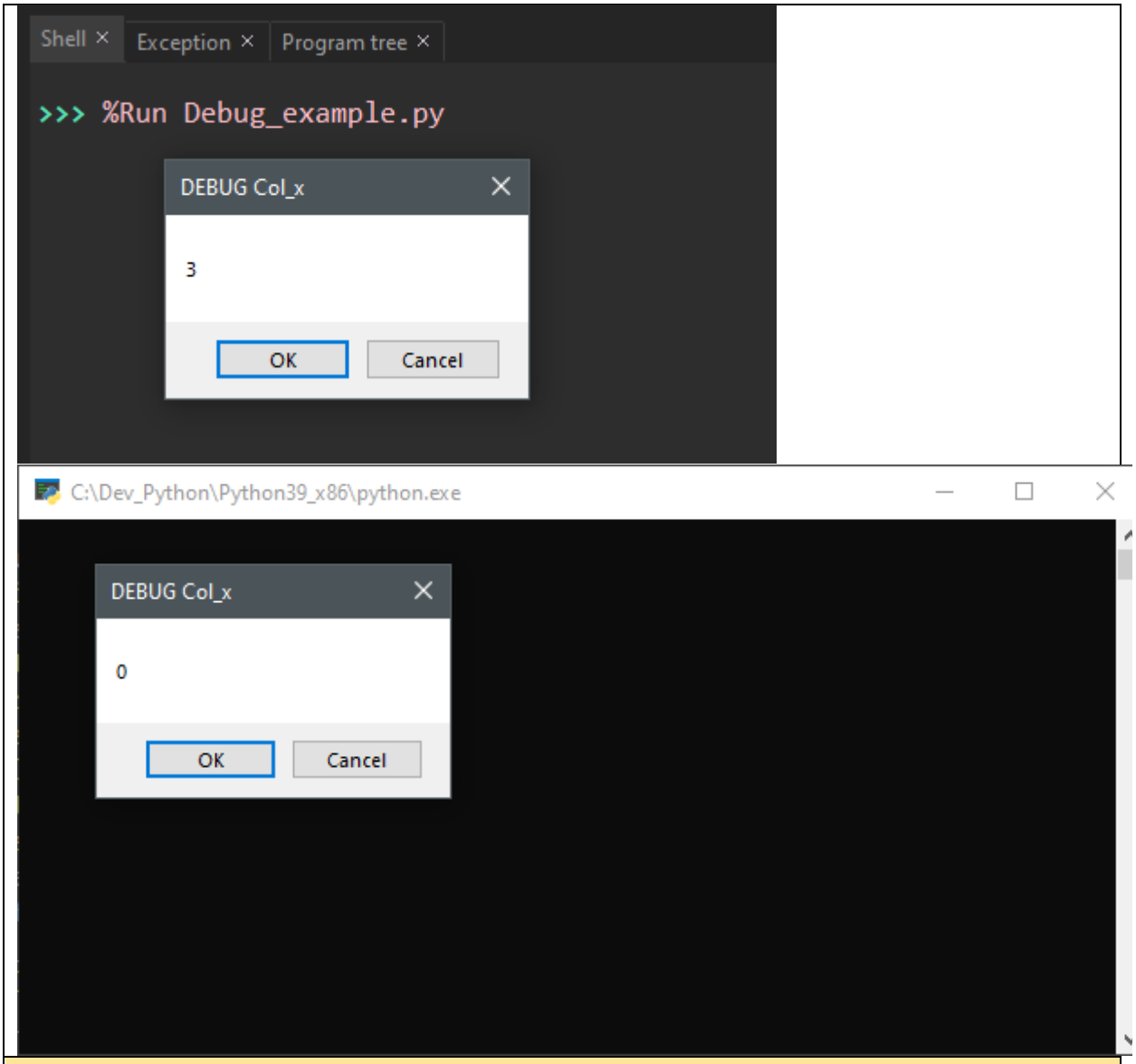
Laguage: Python 3 – CLI/GUI
Code example: “Debug_example.py”

```
# Print a table 5 rows x 5 columns
# with each column numbered 6 to 10
def main():
    ROWS = 5
    COLUMNS = 5
    Row_y = 0
    Col_x = 0
    MyArray = [[None]* COLUMNS for _ in range(ROWS)]

    for Row_y in range(0,  ROWS):
```

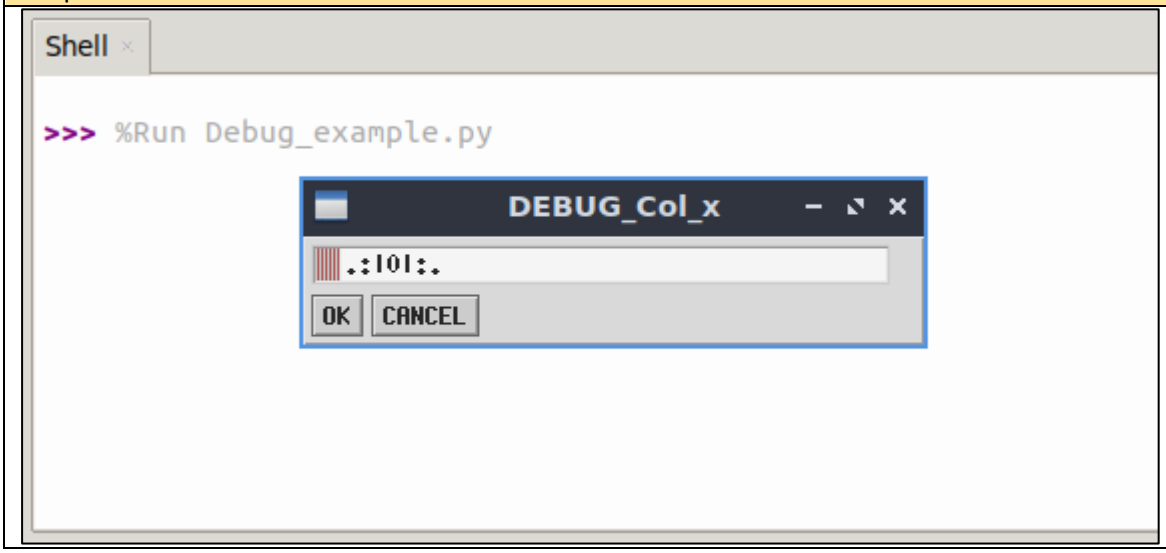
```
        for Col_x in range(0,  COLUMNS):
            MyArray[Row_y][Col_x] = Col_x +5
            DebugMsg("DEBUG_Col_x", str(Col_x))

    for Row_y in range(0,  ROWS):
        for Col_x in range(0,  COLUMNS):
            print(MyArray[Row_y][Col_x], end="")
        print("")

    input("Press [Enter] to exit.")
    return None

# A simple message box for debugging.
# Takes a String value only, so you will have to convert numbers to String.
def DebugMsg(aTitle, aMessage ):

    import sys
    #import os

    # for windows
    if sys.platform.startswith('win32'):
        import win32api
        reterr = 0
        # MessageBox[W] is for Unicode text, [A] is for ANSI text.
        ## Alternative
        #import ctypes
        #ctypes.windll.user32.MessageBoxW(0, aMessage, aTitle, 0)
        # To attach to the parent console window.
        #MessageBoxA(FindWindowA("ConsoleWindowClass", NULL),msg,title,0);
        reterr = win32api.MessageBox(0, aMessage, aTitle, 1)  # 65536 =
MB_SETFOREGROUND
        if(reterr == 0):  # Holds the return value from the command line.
            # MessageBox() Fail
            return 0
        elif(reterr == 1):
            # IDOK
            return 1
        elif(reterr == 2):
            # IDCANCEL (Ctrl + C is dissabled when GUI messagebox is used)
            # So provide an option to break out of the debugging...
            # This will also exit now if the Close X is selected.
            sys.exit(0)  # Clean exit the application.
        else:
            return -1;

        return 0
    # for linux
    elif sys.platform.startswith('linux'):
        import os
        # http://manpages.ubuntu.com/manpages/trusty/man1/xmessage.1.html
        # apt-get install x11-utils
        #system("xmessage -center 'Hello, World!'");
        # Else try wayland
        # https://github.com/Tarnyko/wlmessage
        #system("wlmessage 'Hello, World!'");
        reterr = 0
        Buffer = ""
        Buf_Msg = ""
        # Place title text in 'apostrophe'.
```

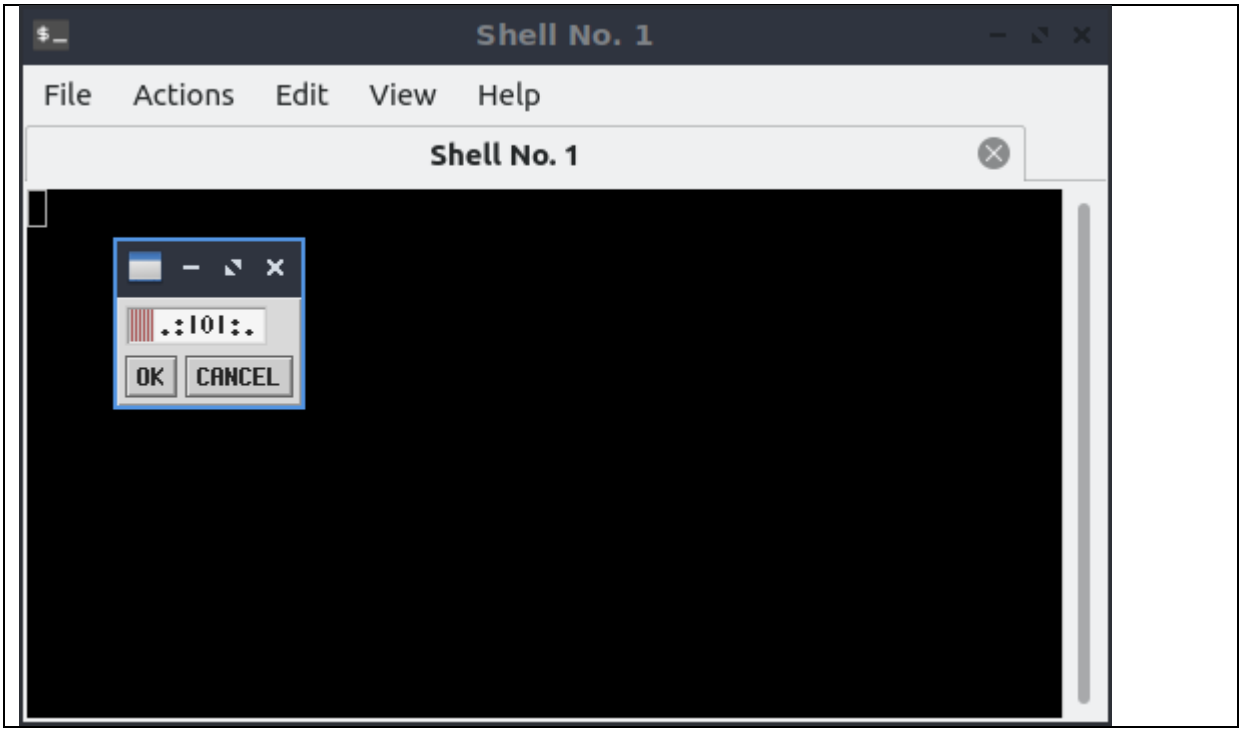
```
        Buf_Msg = "\'" + aMessage + "\'"

        # Build our command line statement.
        # xmessage [-options] [message ...]
        # NOTE! ">>/dev/null 2>>/dev/null" suppresses the console output.
        Buffer = "xmessage -buttons OK:101,CANCEL:102 -center -title " + Buf_Msg +
" \'.:|" + aMessage + "|:.\' >>/dev/null 2>>/dev/null"

        # Send it to the command line.
        reterr = os.system(Buffer)
        # This is the recommended linux way of getting the return from calling
        # a command line application. It has the overhead of including import os
        # under the Linux OS test...
        #if ( os.WIFEXITED(reterr))
            #print("The return value: " + os.WEXITSTATUS(reterr))
        #elif (os.WIFSIGNALED(reterr))
            #print("The program exited because of signal no: " +
os.WTERMSIG(reterr))

        # As the return value is stored in the "High Byte" of the 16bit error
return,
        # we can use some bitwise shift magic to extract the value of the 8 bit
high
        # byte and eliminate the need for the import os library file.
        if(reterr>>8 == 101):
            # ID OK
            pass
        elif(reterr>>8 == 102):
            # ID CANCEL
            sys.exit(0)  # Quit debugging
        elif(reterr>>8 == 1):
            # ID CLOSE X
            pass
        elif(reterr>>8 == 0):
            # ID Default run OK
            pass
        else:  # xmessage failed or not exist.
            # Try Wayland compositor wlmessage.
            Buffer = "wlmessage \' |" + aMessage + "| \' >>/dev/null 2>>/dev/null"
            reterr = os.system(Buffer)
            if(reterr>>8 != 0) & (reterr>>8 != 1):
                # Popup message failed.
                #printf("%d\n", reterr);
                return -1

        return 0
    else:
        pass
        return -1  # Other OS
    return None

if __name__ == '__main__':
    main()
```

Output: Windows



Output: Linux





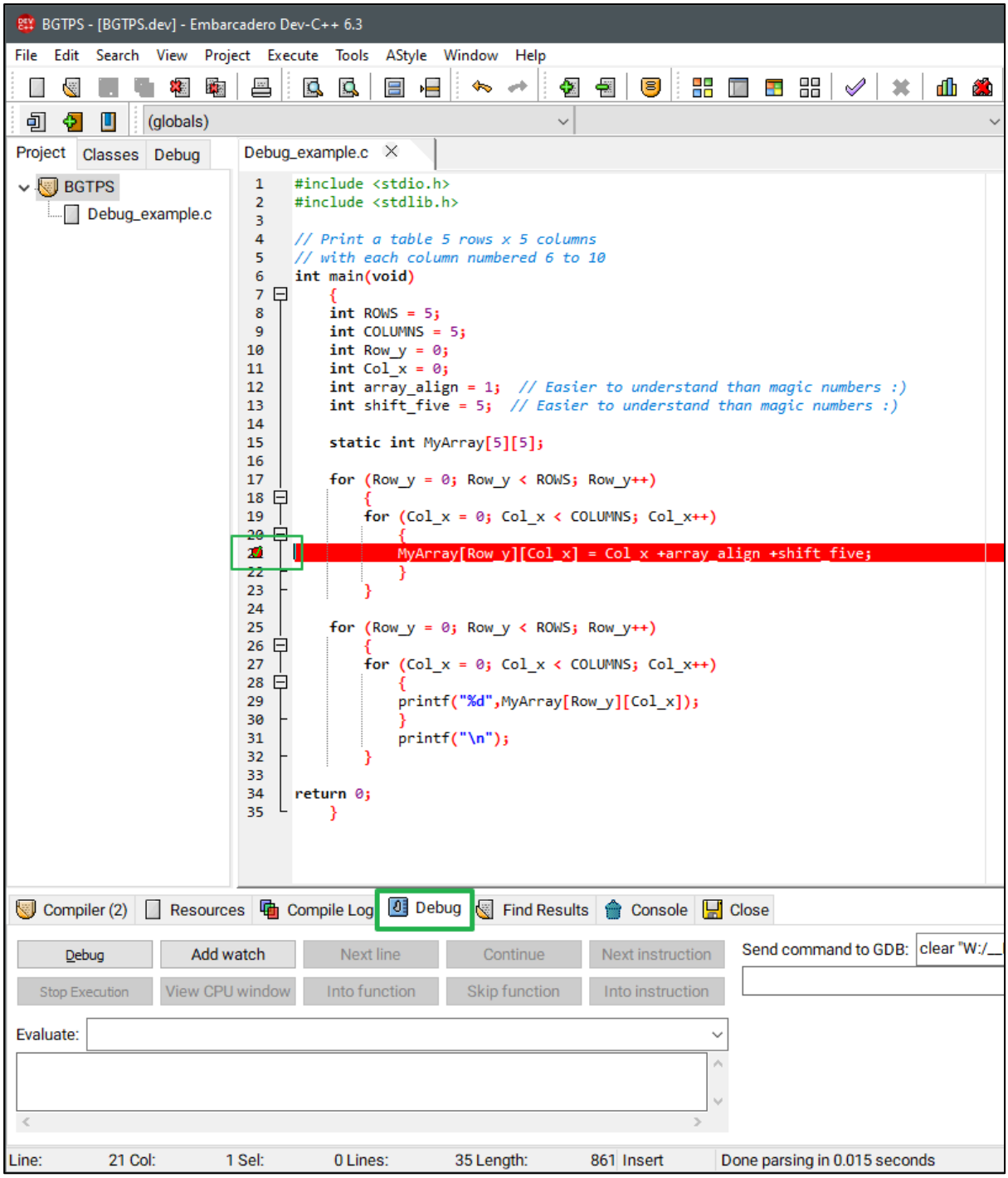
Break Points and watches

Many IDEs allow us to set “Breakpoints” in the margin beside our code. We can run the application up until the point of the breakpoint and make some checks and analysis of our code and variable/s at that point. We can also add a tracer to a particular variable and “Step” through our code monitoring the values of the variable/s.

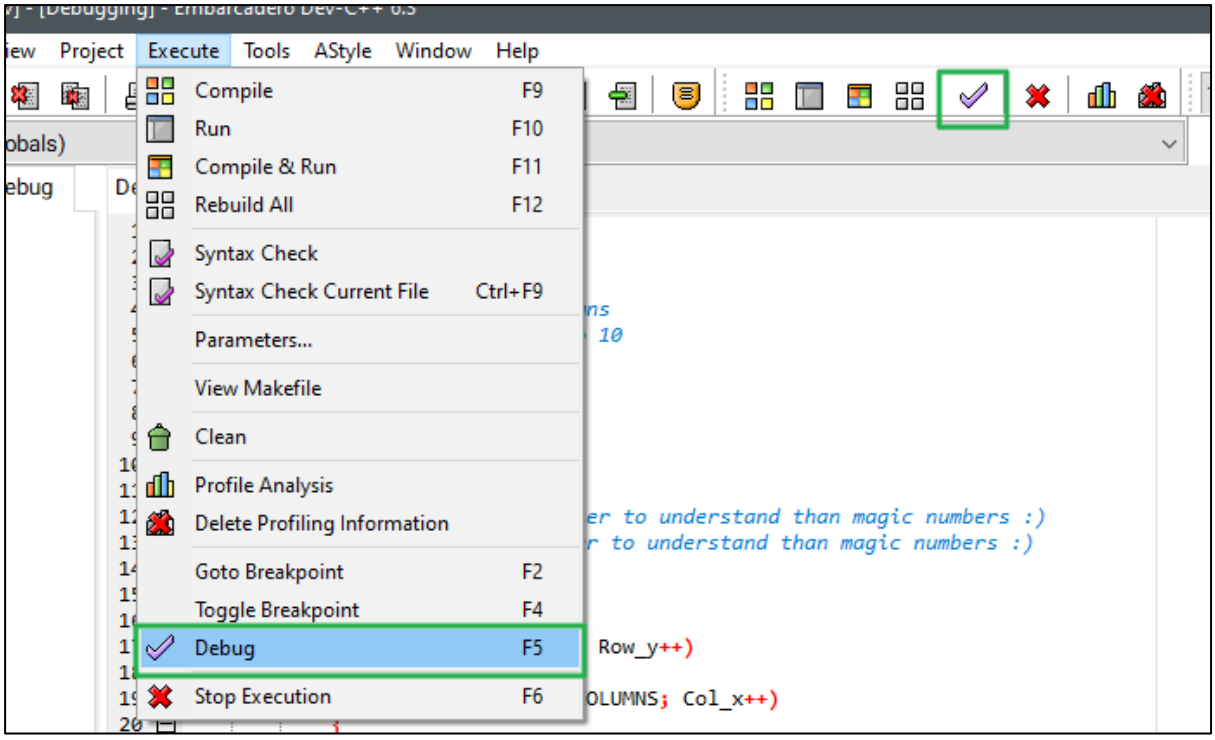
I am only going to offer a brief explanation of using breakpoints and tracers as they are both language specific and IDE specific as well as too complex in range to offer a simple explanation within this booklet. If you look into the language and IDE documentation you will find specific tutorials.

C Dev-C++

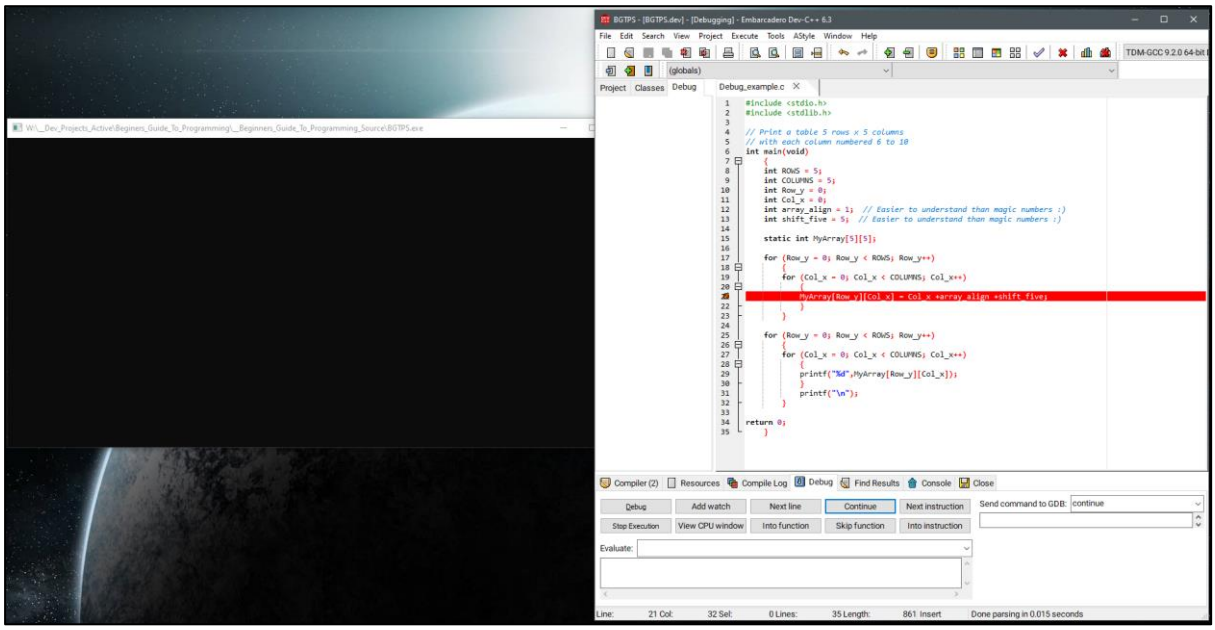
To use Breakpoints and variable watches in Dev-C++ select the **[Debug] TAB**, and then click in the left margin on the line where you want to enable a breakpoint (The application will run in the debugger and pause execution each time that it reaches this point).



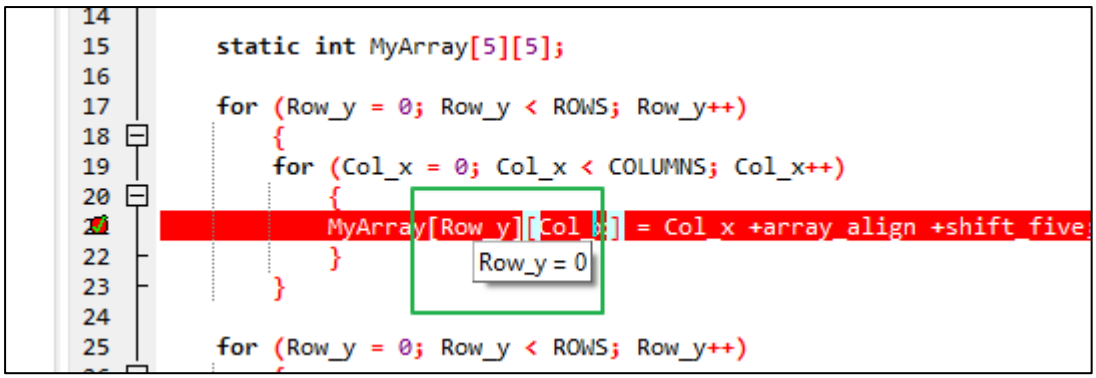
Next Select “Debug F5” to run the debugger to the break point. You can also use the [Debug] button in the lower Debug TAB.

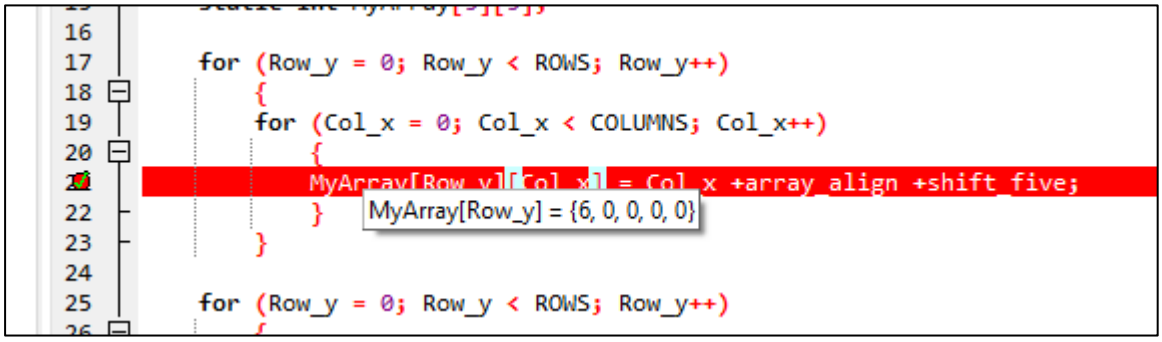


A command console window will pop up the same as when you “Run” the application. You will need to move the console window to the side of the IDE so that you can see both.

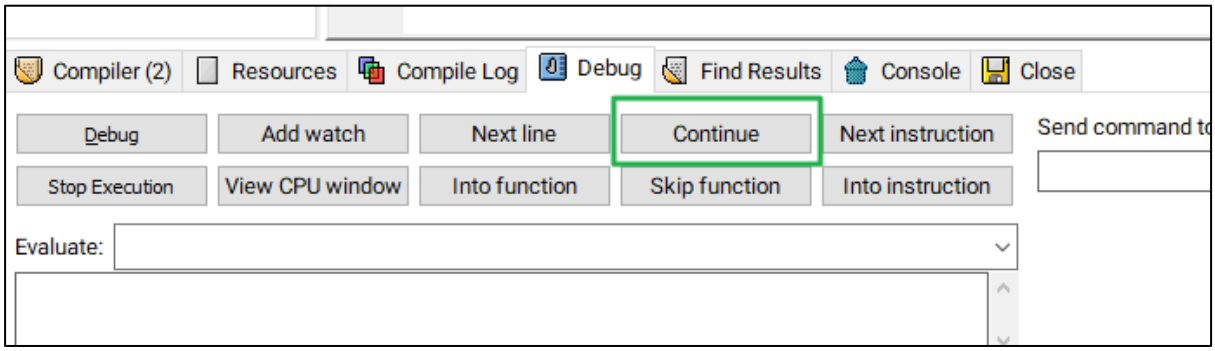


Next, hover your mouse pointer over any variable including arrays and you will see an information pop up showing the current value for the variable.





Select the [Continue] button to continue execution to the next break point.

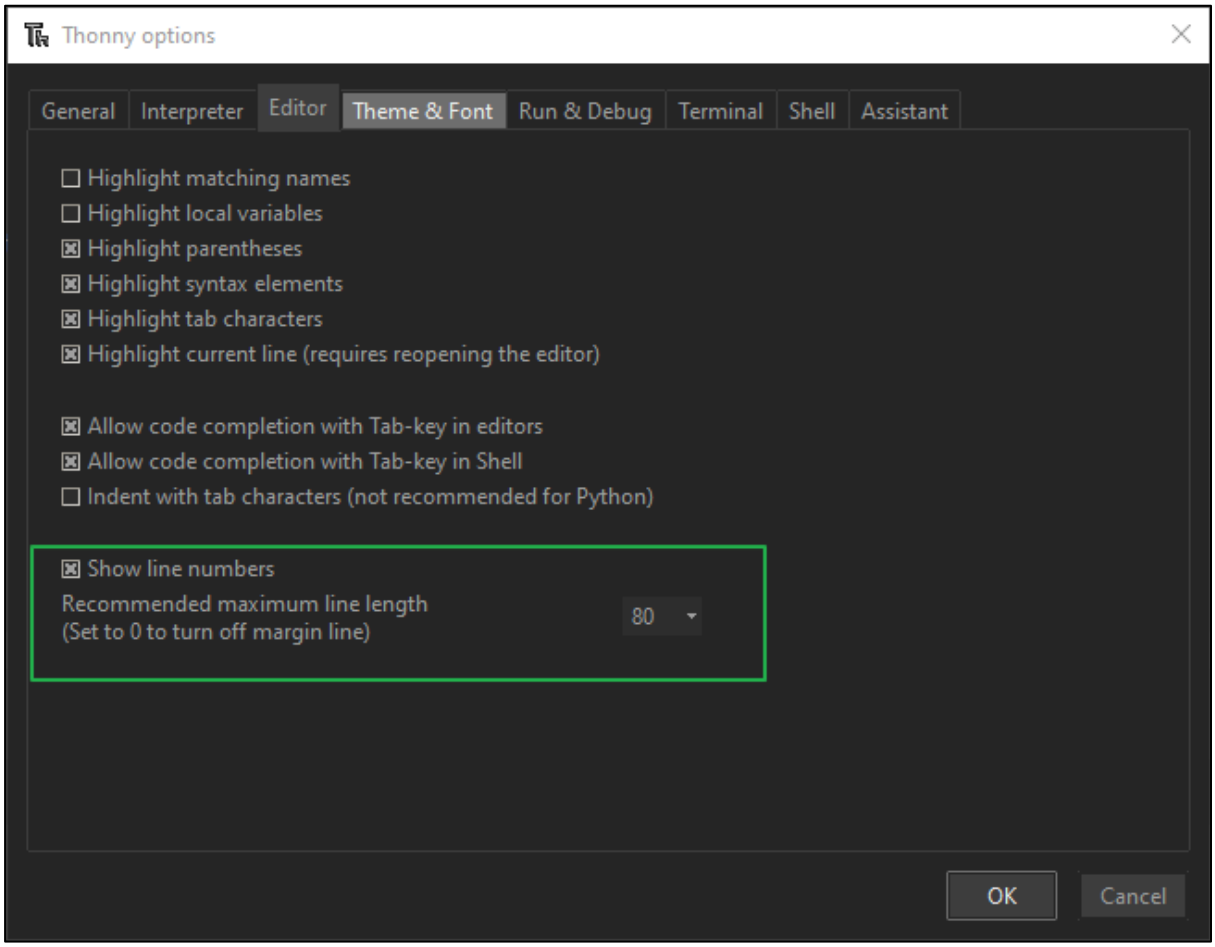


You can repeatedly select continue, and then hover over each variable that would like to see the value of each time the execution pauses at the break point. You can select the [Stop Execution] button to terminate the debug process.

Python 3 Thonny

To use Breakpoints and variable watches in Thonny, ensure that line numbers have been enabled.

Go to Tools -> Options… -> Editor TAB
Select [X] Show Line Numbers, and then select [OK]

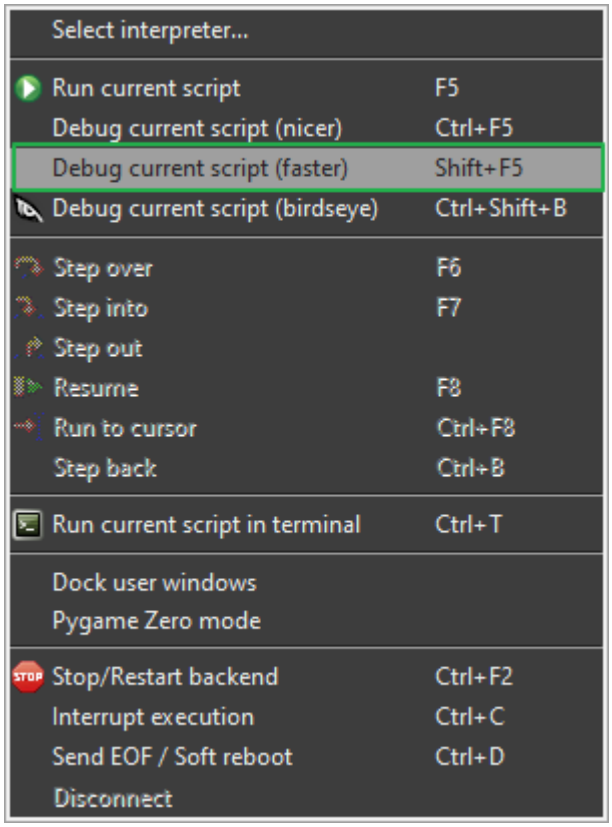


Double click on the line number that you would like the application to “Break” at as well as show the Variable watch values for.

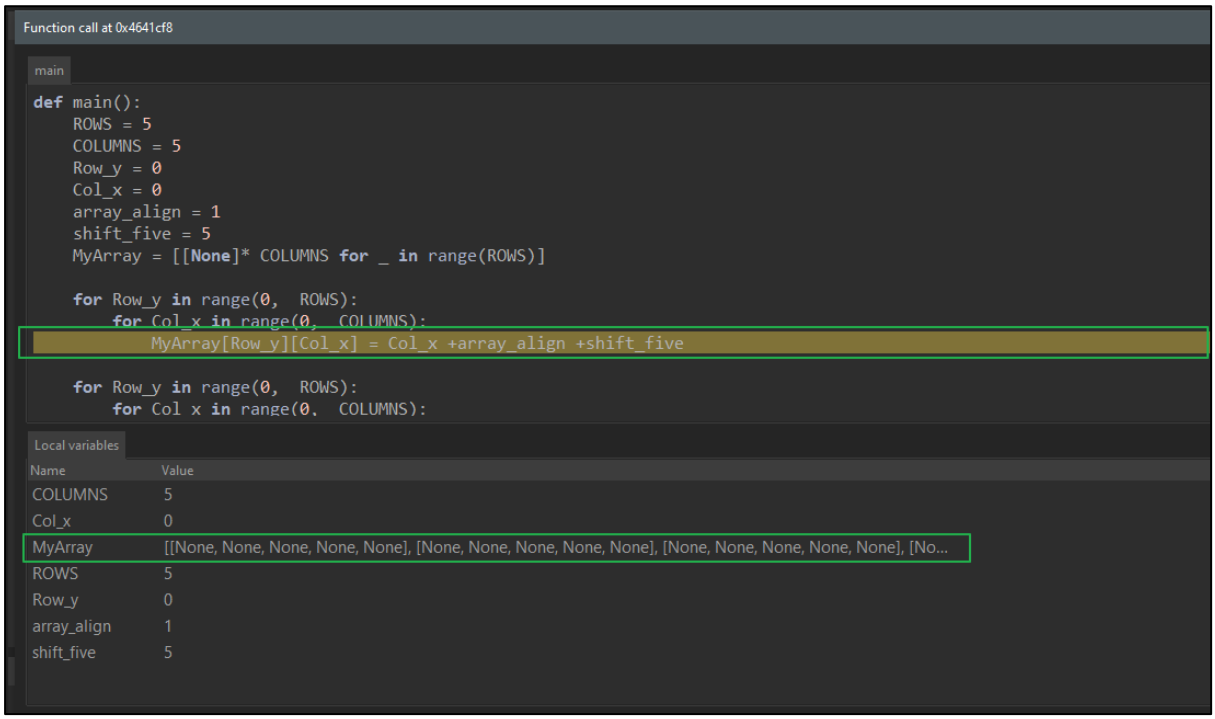
Debug_example.py

```
# Print a table 5 rows x 5 columns
# with each column numbered 6 to 10
def main():
    ROWS = 5
    COLUMNS = 5
    Row_y = 0
    Col_x = 0
    array_align = 1
    shift_five = 5
    MyArray = [[None]* COLUMNS for _ in range(ROWS)]

    for Row_y in range(0,  ROWS):
        for Col_x in range(0,  COLUMNS):
            MyArray[Row_y][Col_x] = Col_x +array_align +shift_five

    for Row_y in range(0,  ROWS):
        for Col_x in range(0,  COLUMNS):
            print(MyArray[Row_y][Col_x], end="")
        print("")

    input("Press [Enter] to exit.")
    return None

if __name__ == '__main__':
    main()
```

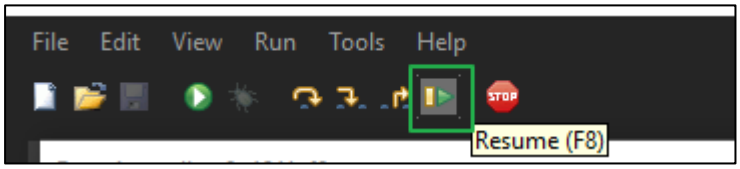
To run the application to the first break point select Run -> Debug Current Script (Faster) Shift + F5.

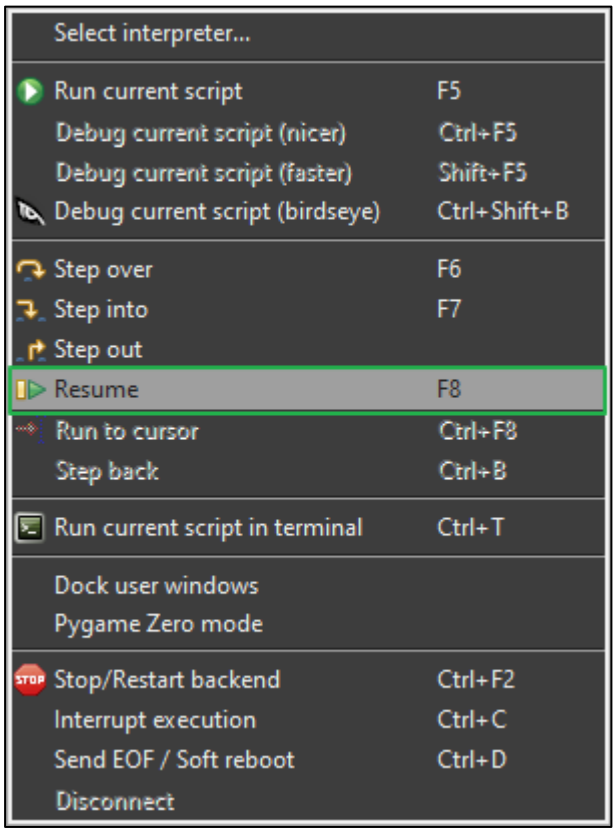


A new TAB will emerge below your script showing all of the variable values for that line at the current break in the program execution.

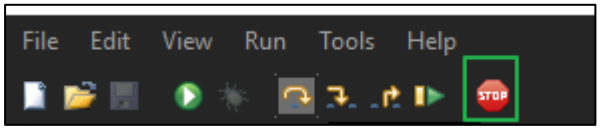


To continue to the next breakpoint, select Resume F8. This will allow you to step through the application's execution pausing at each "Breakpoint" to "Watch" the changes in value of each variable.





You can continue to select Resume until the application ends or if you can see the problem with your source you can end the debug session by selecting Stop.



Unit Tests

It is often beneficial to copy the problematic section of code into a source document on its own and treat it as a separate application. This can be particularly useful in large complex source code as we can isolate the block of routines making it simpler and easier to run without all the extra overhead of a full application. This also makes it easy to apply unit tests to the individual routine so that we can test the full range of expected outcomes for erroneous results. This is also a major reason for

creating applications in a modular way using functions as we can easily test the function outside of the main application and then swap the corrected function in without breaking our main source.

I usually create routines and functions separately from my main source. This allows me to fully bug test the routines in a more simplified environment prior to including them into the main source. I can usually have a high degree of confidence that the routines will work as expected and any errors may just be in passing the data to the routines.

Error and Exception Handling.

Exception handling is the practice of dealing with unpredictable data inputs into our application. There are many instances where a file may not exist or a user may enter data that is outside of the expectations of our software. As programmers and coders we are required to "Predict" and occasion where this could occur and create "Boundaries" that capture and handle programs flow if it goes outside of the expected boundaries.

"Error and Exception Handling" and "Code safety and security" tend to blend in practice.

Most modern programing modules, functions and objects as well as applications will return some form of error code when something does not go as expected. In most cases it is up to the programmer to create and make use of error checking routines and logic within their application design. It is also up to the application designer to correctly handle errors in their application and take the necessary steps to correct the error, or in the worst cases issue a warning to the user and "Gracefully" shut down the application. The most serious of errors will be caught by modern operating systems and also perform a "Less than graceful" shutdown of the application.

At any point in an application that you retrieve data from a source external to the application, for example a database file, input from a user, or data downloaded from the internet there is a high risk that the application will not receive the data that is expected. In each of these situations we must add some form of true false comparison and check that the data is within the expected range required. If not we can make further attempts to retrieve the correct information, or issue a warning to the user, for example "file Config.csv not found", or "The user's input has exceeded the 64 character limit".

There are many different ways to achieve this in practice and at times can be as complicated as creating the original "Functional design" of an application.

"It is easy to write an application to perform the tasks you want it to do. The challenging part is stopping your application from doing the tasks that you don't want it to do."
Axle

Although I have not included a great deal of error checking in the examples of this booklet in an attempt to keep the program flow and logic simplified and easier for a beginner to follow You will find that I have included some error checking where I have written to or read from a "File Open" routine. Another example is the routines that check for the operating system in use so that I can make use of the correct library functions. In these cases it was unavoidable to create a functional application without the use of error checks.

A simple error check routine

An example of requesting an input from a user and ensuring that the entered data is safe. Safe meaning that it is within the range or length of the maximum allowed by our buffer (Variable) and that the data is of the type that is expected.

The second part will show an example of checking the error return from a common function call.

Language: C

Code Example: “error_check.c”

```
// C std library headers.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Helper functions
void S_Pause(void);
int S_getchar(void);
// v Emulates Input function in FreeBASIC and Python 3.
char *Input(char *buf, char *str, int n);
char *S_fgets(char *buf, int n, FILE *stream);
// Maximum buffer size (Max length of a string + 1 for '\0')
#define MAX_BUFFER 128  // Set a maximum size for buffers and never exceed it.

int main(void)  // Main procedure
    {
    char Input_Buffer[MAX_BUFFER] = {'\0'};  // buffer
    unsigned int Zero_Passlength = 0;  // No Password entered
    unsigned int Min_Passlength = 6;  // Pass length Min limit
    unsigned int Max_Passlength = 12;  // Pass length Max limit
    unsigned int Max_Attempts = 3;  // Attempts limit
    unsigned int Attempts_Counter = 0;  // Attempts count
    unsigned int Opt_Out = 0;  // Opt out question y/n

    // Test that the input data is within the expected limits of 6 to 12.
    // Loops until the user enters the correct data. In real life we would
    // also need an opt out after so many tries.
    // This may appear like a lot of extra code, but it is necessary in C for
    // safety. Unlike many "High Level" languages C does not have built in
    // safeguards so it is up to the coder to be pandantic about safety.
    // If you remove all of my "Comment explanations" you will find it is not
    // that much extra code :)
    // Note. C++, BASIC and Python have far more built in buffer safeguards,
    // but it is still up to the coder to ensure that any external input is
    // within the expected range of data.
    while(strlen(Input_Buffer) < Min_Passlength
        || strlen(Input_Buffer) > Max_Passlength)
        {
        // The next conditional check is part of an unwind to break out of deeply
        // nested loops. Although there are other methods to do this as well, I
        // wanted to keep a simple method arcoss all 3 languages.
        // The above "While" conditional test should break the loop if the
        // Password is the correct length without the folowing test, but I wanted
        // to show an unwind method just the same :)
        if(Opt_Out == 'Y' || Opt_Out == 'y')
            {
```

```
            break;  // Quit asking for a name and break out of the loop "Step 2".
            // If we wanted to reuse the Opt_Out variable again later I suggest
            // uncommenting the following line.
            //Opt_Out = 0;  // Reset Y/N the response variable.
            }
        else
            {
            // Note that we have 2 safety checks for the string that can be
entered.
            // the first is the MAX_BUFFER which limits all input to be less
            // than the buffer "n" Input( , , int n). this blocks the possability
            // of a buffer overflow if the user inputs a longer string than the
            // buffer can hold. See the comments in Input().
            Input(Input_Buffer, "Please enter your password.\n Between 6 to 12
letters:"
            , MAX_BUFFER -1);
            // The second part of our test is to see if the data is within the
            // range that is expected.
            if(strlen(Input_Buffer) == Zero_Passlength)  // 0 length string.
                {
                printf("You did not enter your password...\n");
                Attempts_Counter++;
                }
            else if((strlen(Input_Buffer) > Zero_Passlength)
                 && (strlen(Input_Buffer) < Min_Passlength))  // String shorter
than 6.
                {
                printf("The password you entered is too short...\n");
                Attempts_Counter++;
                }
            else if(strlen(Input_Buffer) > Max_Passlength)  // String longer than
12.
                {
                printf("The password you entered is too long...\n");
                Attempts_Counter++;
                }
            else  // String is withing range. Success.
                {
                printf("Your Password is %s\n", Input_Buffer);
                }

            // Limit the number of attempts and offer an opt out so that the
            // user is not caught in an endless loop if they decide to not
            // enter a name.
            if(Attempts_Counter >= Max_Attempts)
                {
                // keep asking in the loop untill we get a valid Y/N response.
                while(Opt_Out != 'y' && Opt_Out != 'Y')
                    {
                    printf("\nSorry you have reached the maximum number of
tries!\n");
                    printf("Would you like to quit? (Y/N):\n");
                    // Characters are unsigned integers in C: int Ch = 'Y' =
(char)89;
                    // Strings are an array of characters char St = "Y"; = pionter
                    // to St[0] = (char)89 = (char)'Y'
                    Opt_Out =  S_getchar();
                    if(Opt_Out =='y' || Opt_Out == 'Y')
                        {
```

```
                        break;  // Quit asking for a name and break out "step 1".
                        }
                    else if(Opt_Out == 'n' || Opt_Out == 'N')
                        {
                        Attempts_Counter = 0;  // reset the attempts counter
                        Opt_Out = 0;  // reset the opt out counter
                        break;
                        }
                    else
                        {
                        // ask again until we get a Y/N response.
                        printf("Invalid response!\n");
                        }
                    }
                }
            }

        }

    printf("\nFile read error test\n");
    // Checking the error return of a function. This is always recomended when
    // the function handles data from an unknown source aka anything outside
    // of the source code of you application. This icludes "User Inputs",
    // "Data from a file or database", "Information from the web",
    // "Communication and data transfers to other apps" ++.
    // We can never guarentee the existance of data outside of our application
    // or if it will be the data that we have expected.
    //
    // Description of:
    // FILE *fopen(const char *filename, const char *mode)
    // Description
    // The C library function FILE *fopen(const char *filename, const char
    // *mode) opens the filename pointed to, by filename using the given mode.
    // Parameters:
    // filename -- This is the C string containing the name of the file to be
    // opened.
    // mode -- This is the C string containing a file access mode. It includes:
    // ...
    // Return Value: (<- this is what you need to look for.)
    // This function returns a FILE pointer. Otherwise, NULL is returned and the
    // global variable errno is set to indicate the error.
    // If it fails to open because the file does not exist for example, we need
    // to handle the error and either create the file, or tell the user that the
    // file could not be found, or any other number of options that are apropriate
    // to the context of your application. Don't ever let and error be passed to
    // your user with the horrible "Ding Sound" and the
    // "This program has terminated unexpectedly!" warning in production code. :)
    char *filename = "filename.txt";  // a dummy file name.
    FILE *fp;
    fp = fopen(filename, "r");
    // in this example "filename.txt" does not exist so the variable fp will be
    // sent an error as the return "NULL". (Null pointer is a special form of
'0').
    // Yes it sounds weird, but computers have many types of Zero and not all
    // are equal NULL != null != 0 != -0 != FALSE; but are similar.
    if(fp == NULL)
        {
        // Please note that attempting to display a FILE* as an int %d is
        // classed as "undifined behavior" and will throw a compiler warning. You
```

```
        // could also use %p for void pointer, but that too is undefined.
        // I have only done this to show that it will display 0 but should not
        // be used or relied upon in production code. Always compare FILE* to
        // NULL. NULL is a special MACRO that represents the Zero value, or the
        // value of memory of a location of FILE * (pointer) to a structure that
        // does not exist.
        // if(fp == NULL) <- correct | if(fp == 0) <- undefined behavior.
        printf("ERROR! Cannot open file %s\n", filename);
        printf("fp * = %d\n", fp);
        // perror() will retreive the error sent to the console "stderr" and
        // print the value to "stdout".
        perror("Error in opening file");
        // Do some error handling tasks to deal with why the file does not exist.
        // Maybe you need to create the file first?
        // If this is a function that you have created you may wish to "return"
        // some usefull information to the function call.
        //return -1;  // -1 Indicates an error on some platforms.
        }
    else
        {
        // No errors, so do some file read operations.
        }
    fclose(fp);  // Always close files when finished, always.

    // It is common in C to use getchar() as a pause to wait for keybord [Enter]
    // to continue. If more than 1 character is entered then the remaining
    // characters are left in the keyboard buffer and appear next time the
    // keyboard buffer is read. including the new line '\n' for enter.
    // To eleiminate this we have to clear the keyboard buffer before or after
    // each entry. My function S_getchar() "Safe Get Character" enumerates all
    // characters in the buffer clearing any unused keyboard scan codes. The
    // next tim we call for a keyboard input there are no leftover artifacts in
    // the buffer :)
    // A another alternative than this is my Input() or S_fgets()
    // "Safe Get String"  based upon getc() "Get Char" which allows us to enter
    // a maximum nuber of characters to be retreived from stdin to our buffer.
    // S_getchar() returns a single char as int, whereas S_fgets() "Safe Get
String"
    // returns a char array (String) of maximum length n.
    S_Pause();
    return 0;
    }  // END main() <---

// --> START User defined functions and wrappers

// Safe Pause. A "Pause" wrapper for S_getchar().
void S_Pause(void)
    {
    // This function is referred to as a wrapper for S_getchar()
    printf("\nPress [Enter] to continue...");
    S_getchar();  // Uses S_getchar() for safety.
    }

// Safe getcar() removes all artefacts from the stdin buffer. This function
// is how we must always retreive keyboard single character input using a loop
// to enumerate all characters in the stdin buffer keeping only the first
// character and discarding any other characters including the '\n' [Enter].
// This finction is a wrapper that reduces the boiler plate code to a single
// line so we dont have to write the same loop every time we ask for a character
```

```
// from the keyboard. Modular function libraries are the key to convenient
// coding in C. C++ (High level and OOP) includes many of these libraries by
// default. C++ is a more convenient and safer upgrade to writting in C.
int S_getchar(void)
    {
    // This function is referred to as a wrapper for getchar()
    int i = 0;
    int ret;
    int ch;
    // The following enumerates all characters in the buffer.
    while((ch = getchar()) != '\n' && ch != EOF )
        {
        // But only keeps and returns the first char.
        if (i < 1)
            {
            ret = ch;
            }
        i++;
        }
    return ret;
    }

// Console Input (stdin).
// This a a wrapper function that emulates the "Input()" function found
// in BASIC and Python 3.
// buf is the return buffer, str is printed to the screen.
// returns a pointer to buf, str is the input message...
// n is the MAX number of characters allowed. This is considered a safe input
// function as it limits the length of the input string to n reducing the risk
// of a buffer overflow.
// buf must be at least n + 1 for '\0'
char *Input(char *buf, char *str, int n)
    {
    FILE *stream = stdin;
    char *empty = "";
    int ret;
    // Test if an empty string has been sent. If so...
    ret = strcmp(str, empty);
    if(ret != 0)  // Don't print an empty string.
        {
        printf("%s", str);  // Otherwise, Print the Input Message.
        }
    return S_fgets(buf, n, stream);  // Call my safe get string function.
    // and passes the truncated string back to the original function call.
    }

// Safe fgets() removes all artefacts from the stdin buffer.
// buf must be at least n + 1 for '\0'
// Returns a pointer to char array (String) of length n.
char *S_fgets(char *buf, int n, FILE *stream)
    {
    int i = 0;
    int ch;
    //memset(buf, 0, n);
    // The following enumerates all characters in the keybord buffer.
    // The while loop will exit if it encounters an [Enter] '\n' or
    // the end of the file/stream EOF.
    while((ch = getc(stream)) != '\n' && ch != EOF )
        {
```

```
        // But only keeps and returns n chars.
        // This is my safe string feature that only returns n charaters length
        // in the returned string. No buffer overflows for me :)
        // All additional charcters entered by the user are discarded.
        // It should be noted that the coder should promt the user to
        // "Enter a MAX of n characters".
        if (i < n)
            {
            buf[i] = ch;
            }
        i++;
        }
    buf[i] = '\0';
    return buf;  // returns the truncated string.
    }
```

Language: FReeBASIC

Code Example: “error_check.bas”

```
Declare Function main_procedure() As Integer
Declare Function Con_Pause() As Integer  ' GetKey Version
main_procedure()

Function main_procedure() As Integer  ' Main procedure
    ' the MAX size of a String type in FB is 2GiB and based upon "Dynamic Memory".
    ' We CAN set a "Static Memmory" size with MyString(128) but is generally
    ' not recomended. https://documentation.help/FreeBASIC/TblVarTypes.html
    Dim As String Input_Buffer = ""
    Dim As UInteger Zero_Passlength = 0  ' No Password entered
    Dim As UInteger Min_Passlength = 6  ' Pass Min length limit
    Dim As UInteger Max_Passlength = 12  ' Pass Max length limit
    Dim As UInteger Max_Attempts = 3  ' Attempts limit
    Dim As UInteger Attempts_Counter = 0  ' Attempts count
    Dim As String Opt_Out = ""  ' Opt out question y/n

    ' Test that the input data is within the expected limits of 6 to 12.
    ' Loops until the user enters the correct data. In real life we would
    ' also need an opt out after so many tries.
    ' This may appear like a lot of extra code, but it is necessary to keep a
    ' level of safety.
    ' If you remove all of my "Comment explanations" you will find it is not
    ' that much extra code :)
    ' C++, BASIC and Python have far more built in buffer
    ' safeguards than C, but it is still up to the coder to ensure that any
    ' external input is within the expected range of data.
    While(Len(Input_Buffer) < Min_Passlength) _
        Or (Len(Input_Buffer) > Max_Passlength)
        ' The next conditional check is part of an unwind to break out of deeply
        ' nested loops. Although there are other methods to do this as well, I
        ' wanted to keep a simple method arcoss all 3 languages.
        ' The above "While" conditional test should break the loop if the Password
        ' is the correct length without the folowing test, but I wanted to show
        ' an unwind method just the same :)
        If(Opt_Out = "Y") or (Opt_Out = "y") Then
            Exit While  ' Quit asking for a name and break out of the loop "Step
2".
            ' If we wanted to reuse the Opt_Out variable again later I suggest
            ' uncommenting the following line.
```

```
            'Opt_Out = ""  ' Reset Y/N the response variable.
        else
            ' Note that FreeBASIC has built in limit for the input string length
            ' so we only have to check the the return is withing the range and
            ' data type expected.
            Print "Please enter your password."
            Input " Between 6 to 12 letters:", Input_Buffer
            ' The second part of our test is to see if the data is within the
            ' range that is expected.
            If(Len(Input_Buffer) = Zero_Passlength) Then  ' 0 length string.
                Print "You did not enter your password..."
                Attempts_Counter += 1
            ElseIf((Len(Input_Buffer) > Zero_Passlength) _
                And (Len(Input_Buffer) < Min_Passlength)) Then  ' String shorter
than 6.
                Print "The password you entered is too short..."
                Attempts_Counter += 1
            ElseIf(Len(Input_Buffer) > Max_Passlength) Then  ' String longer than
12.
                Print "The password you entered is too long..."
                Attempts_Counter += 1
            Else  ' String is withing range. Success.
                Print "Your Password is " & Input_Buffer
            End If

            ' Limit the number of attempts and offer an opt out so that the
            ' user is not caught in an endless loop if they decide to not
            ' enter a name.
            If(Attempts_Counter >= Max_Attempts) Then
                ' keep asking in the loop untill we get a valid Y/N response.
                While(Opt_Out <> "y") And (Opt_Out <> "Y")
                    Print !"\nSorry you have reached the maximum number of
tries!\n"
                    Print "Would you like to quit? (Y/N):";

                    Input Opt_Out
                    If(Opt_Out = "y") Or (Opt_Out = "Y") Then
                        Exit While  ' Quit asking for a name and break out "step
1".
                    ElseIf(Opt_Out = "n") Or (Opt_Out = "N") Then
                        Attempts_Counter = 0  ' reset the attempts counter
                        Opt_Out = ""  ' reset the opt out counter (3 more tries)
                        Exit While
                    Else
                        ' ask again until we get a Y/N response.
                        Print "Invalid response!"
                    End If
                WEnd
            End If
        End If
    WEnd

    Print ""
    Print "File read error test 1"
    ' Checking the error return of a function. This is always recomended when
    ' the function handles data from an unknown source aka anything outside
    ' of the source code of you application. This icludes "User Inputs",
    ' "Data from a file or database", "Information from the web",
    ' "Communication and data transfers to other apps" ++.
```

```
    ' We can never guarentee the existance of data outside of our application
    ' or if it will be the data that we have expected.
    '
    ' Description of:
    ' result = Open(filename For Input [encoding_type] [lock_type] As
[#]filenumber)
    ' Description
    ' https://documentation.help/FreeBASIC/KeyPgOpen.html
    ' Return Value: (<- this is what you need to look for.)
    ' In the first usage, Open returns zero (0) on success and a non-zero error
    ' code otherwise.
    ' https://documentation.help/FreeBASIC/TblRuntimeErrors.html
    ' 2 - File not found signal
    ' If it fails to open because the file does not exist for example, we need
    ' to handle the error and either create the file, or tell the user that the
    ' file could not be found, or any other number of options that are apropriate
    ' to the context of your application. Don't ever let and error be passed to
    ' your user with the horrible "Ding Sound" and the
    ' "This program has terminated unexpectedly!" warning in production code. :)
    Const filename As String = "filename.txt"  ' a dummy file name.
    Dim As Integer errorvalue = 0  ' Variable to store retured error codes.
    ' In some instances error codes can be retreived with the MACRO Err .
    '==> Open file for text read ops.
    Dim As Long Fp
    Fp = Freefile()  ' Similat to a FILE pointer in C.
    ' in this example "filename.txt" does not exist so Open( ...) will
    ' send an error as the return. A return of 0 means that there were no
    ' errors in opening the file. Any other value indicates an error.
    ' Open file for text read ops.
    errorvalue = Open(filename, For Input, As #Fp)
    If (errorvalue <> 0) Then  'If (Err <> 0) Then  ' Alternative method
        Print "Err = " & Err  ' the use of Err must come before any Print.
        Print "errorvalue = " & errorvalue
        Print "ERROR! Cannot open file " & filename
        Con_Pause() ' Wait until a key is pressed
        ' Do some error handling tasks to deal with why the file does not exist.
        ' Maybe you need to create the file first?
        ' If this is a function that you have created you may wish to "return"
        ' some usefull information to the function call.
        'Return -1  '-1 Indicates an error on some platforms.
    Else
        ' No errors, so do some file read operations.
        Close #Fp  ' Always close files when finished, always.
    End If

    Print ""
    Print "File read error test 2"
    ' Alternative method.
    ' This is also a common error check method in FreeBASIC
    ' Err will report the last runtime error. You must check it, or store the
    ' RetError = Err before any other internal function are called as the error
    ' value will be reset.
    ' Open file for text read ops.
    If Open(filename, For Input, As #Fp) <> 0 Then
        errorvalue = Err  ' Savee a copy of Err as it will be reset with next
Print.
        Print "Err = " & Err  ' The use of Err must come before any Print.
        Print "ERROR! Cannot open file " & filename
        Print "Err of last Print statement = " & Err
```

```
        Print "errorvalue = " & errorvalue
        Con_Pause() ' Wait until a key is pressed
        ' Do some error handling tasks to deal with why the file does not exist.
        ' Maybe you need to create the file first?
        ' If this is a function that you have created you may wish to "return"
        ' some usefull information to the function call.
        'Return -1  '-1 Indicates an error on some platforms.
    Else
        ' No errors, so do some file read operations.
        Close #Fp  ' Always close files when finished, always.
    End If

    ' You will often see the folowing format for opening a file. I recomend
    ' only using the function version as above Open("file.ext", For Input, As #f)
    ' As this method is for QBasic and will throw a runtime error if not
    ' managed and compiled with the correct compiler switches.
    ' There are many different ways to open files for reading and writting in
    ' FreeBASIC.
    ''Dim f As Integer
    ''f = FreeFile
    ''Open "file.ext" For Input As #f
    ''If Err>0 Then Print "Error opening the file":End

    Close  ' Will close ALL open file handles (pointers).

    ' My Console Pause function wrapper uses GetKey function which is safe.
    Con_Pause()
    Return 0
End Function  ' END main_procedure <---

' Console Pause (GetKey version)
' GetKey returns the first key pressed and entered into the keyboard buffer.
' It does not wait for [Return]. The Snancode (Key press) is removed from the
' stdin buffer unlike C where we have to manually loop through the buffer and
' discard additional characters.
' This is just a simple wrapper for GetKey with a print statement. It is
' benificial over time to create your own personal function library for common
' task as it removes the need to type in the 3 lines of code below every time
' you need a pause. If you look at my Input() function from the C example you
' can see that I have reduced some 15 lines of code to a single function call.
' This forms the basic principles for the creation of "Modular Code" and code
' libraries. We can use a library function instead of repeadedly writting
' common ("Boiler Plate") code in our main() source.
Function Con_Pause() As Integer
    Dim As Long dummy
    Print !"\nPress any key to continue..."
    dummy = Getkey
    Return 0
End Function
```

Language: Python3

Code Example: “error_check.py”

```
def main():

    # The MAX size of a String type in Py is limited to the "Dynamic Memory"
    # available to the system. Pythons internals will not aloww the contents
```

```
    # of the input buffer to go above what is available to the system.
    # We can't set a static size for the length (Memory) in Python.
    Input_Buffer = ""
    Zero_Passlength = 0  # No Password entered
    Min_Passlength = 6  # Pass Min length limit
    Max_Passlength = 12  # Pass Max length limit
    Max_Attempts = 3  # Attempts limit
    Attempts_Counter = 0  # Attempts count
    Opt_Out = ""  # Opt out question y/n

    # Test that the input data is within the expected limits of 6 to 12.
    # Loops until the user enters the correct data. In real life we would
    # also need an opt out after so many tries.
    # This may appear like a lot of extra code, but it is necessary to keep a
    # level of safety.
    # If you remove all of my "Comment explanations" you will find it is not
    # that much extra code :)
    # C++, BASIC and Python have far more built in buffer safeguards than C,
    # but it is still up to the coder to ensure that any external input is
    # within the expected range of data.
    while ((len(Input_Buffer) < Min_Passlength) \
        or (len(Input_Buffer) > Max_Passlength)):
        # The next conditional check is part of an unwind to break out of deeply
        # nested loops. Although there are other methods to do this as well, I
        # wanted to keep a simple method arcoss all 3 languages.
        # The above "While" conditional test should break the loop if the Password
        # is the correct length without the folowing test, but I wanted to show
        # an unwind method just the same :)
        if (Opt_Out == "Y") or (Opt_Out == "y"):
            break  # Quit asking for a name and break out of the loop "Step 2".
            # If we wanted to reuse the Opt_Out variable again later I suggest
            # uncommenting the following line.
            #Opt_Out = ""  ' Reset Y/N the response variable.
        else:
            # Note that Python has built in limit for the input string length
            # so we only have to check the the return is withing the range and
            # data type expected.
            print("Please enter your password.")
            Input_Buffer = input(" Between 6 to 12 letters:")
            # The second part of our test is to see if the data is within the
            # range that is expected.
            if (len(Input_Buffer) == Zero_Passlength): # 0 length string.
                print("You did not enter your password...")
                Attempts_Counter += 1
            elif ((len(Input_Buffer) > Zero_Passlength) \
                and (len(Input_Buffer) < Min_Passlength)): # String shorter than
6.
                print("The password you entered is too short...")
                Attempts_Counter += 1
            elif (len(Input_Buffer) > Max_Passlength):  # String longer than 12.
                print("The password you entered is too long...")
                Attempts_Counter += 1
            else:  # String is withing range. Success.
                print("Your Password is " + Input_Buffer)

            # Limit the number of attempts and offer an opt out so that the
            # user is not caught in an endless loop if they decide to not
            # enter a name.
            if (Attempts_Counter >= Max_Attempts):
```

```
                # keep asking in the loop untill we get a valid Y/N response.
                while (Opt_Out != "y") and (Opt_Out != "Y"):
                    print("\nSorry you have reached the maximum number of tries!")
                    Opt_Out = input("Would you like to quit? (Y/N):")
                    if (Opt_Out == "y") or (Opt_Out == "Y"):
                        break  # Quit asking for a name and break out "step 1".
                               # Opt_Out = "Y" will be used in step 2.
                    elif (Opt_Out == "n") or (Opt_Out == "N"):
                        # reset the attempts counter (3 more tries).
                        Attempts_Counter = 0
                        Opt_Out = ""  # reset the opt out counter.
                        break
                    else:
                        # ask again until we get a Y/N response.
                        print("Invalid response!")

    print("")
    print("File read error test")
    # Checking the error return of a function. This is always recomended when
    # the function handles data from an unknown source aka anything outside
    # of the source code of you application. This icludes "User Inputs",
    # "Data from a file or database", "Information from the web",
    # "Communication and data transfers to other apps" ++.
    # We can never guarentee the existance of data outside of our application
    # or if it will be the data that we have expected.
    #
    # Description of:
    # open(file, mode='r', buffering=-1, encoding=None, errors=None, \
    # newline=None, closefd=True, opener=None)
    # Description
    # https://docs.python.org/3/library/functions.html#open
    # Return Value:
    # Python 3 only returns an "Object" and no error values like C and FreeBASIC.
    # If module core throws an error it is called an exception and the application
    # will terminate if the exception has not been handled. For this we use the
    # try: except method to "Catch" the exception and make a decission on how
    # to manage the error.
    # https://www.w3schools.com/python/python_try_except.asp
    # https://docs.python.org/3/tutorial/errors.html
    # https://docs.python.org/3/library/exceptions.html
    # If it fails to open because the file does not exist for example, we need
    # to handle the error and either create the file, or tell the user that the
    # file could not be found, or any other number of options that are apropriate
    # to the context of your application. Don't ever let and error be passed to
    # your user with the horrible "Ding Sound" and the
    # "This program has terminated unexpectedly!" warning in production code. :)
    # in this example "filename.txt" does not exist so open( ...) will
    # create an exception error FileNotFoundError(2, 'No such file or directory')
    # It is possible that the file may not yet exist. Opening it
    # as "r" will return an exception.
    filename = "filename.txt"  # a dummy file name.
    try:
        # NOTE! With open will automaticaly close the file handle fp so we
        # don't need to use close(fp).
        with open(filename, "r") as fp:  # Open the File.
            # No errors, so do some file read operations.
            Con_Pause()  # wait until a key is pressed

    except FileNotFoundError as e:  # Handle the exception error.
```

```
        print("\nERROR! Cannot open file " + filename)
        print("Maybe the file has not yet been created.")
        # 5 differnt way of retreiving the error message.
        # With e.args[0] we can retreive the error number for some functions.
        # it is not possible to list all the exception types and errors for
        # Python 3 so we usually attempt to cover the most common.
        # except Exception as e: # Will polulate e with any exception class
        # caught with except. You will need to break down the sublevels of each
        # child class to show the actual error such as "FileNotFoundError"
        print(e)
        print(e.args)
        print(e.args[0], e.args[1])
        print(repr(e))
        print(f"{type(e).__name__} at line {e.__traceback__.tb_lineno} of
{__file__}: {e}")
        Con_Pause()

    Con_Pause()  # DEBUG Pause
    return None
    # END Main() <---

# Console Pause wrapper.
# This is just a simple wrapper for Input(). It is
# benificial over time to create your own personal function library for common
# task as it removes the need to type in the 3 lines of code below every time
# you need a pause. If you look at my Input() function from the C example you
# can see that I have reduced some 15 lines of code to a single function call.
# This forms the basic principles for the creation of "Modular Code" and code
# libraries. We can use a library function instead of repeadedly writting
# common ("Boiler Plate") code in our main() source.
def Con_Pause():
    dummy = ""
    print("")
    dummy = input("Press [Enter] key to continue...")
    return None

if __name__ == '__main__':
    main()
```

Code safety and security

Writing secure and safe code is critical in importance once we move into production coding. We have all heard terms like “Zero Day”, “Exploit” and “Flood”. Writing secure code can be challenging, but there are simple steps and methods that will lead to secure code outcomes.

“Error and Exception Handling” and “Code safety and security” tend to blend in practice.

The most common code security vulnerability comes from what is known as a *Buffer Overflow* or *Buffer Overrun*. This occurs when data is allowed to be read or written beyond the defined length of a section of memory established in an array. Some languages such as Python may offer a degree of protection with the use of defined limits on array sizes, but this should never be taken for granted. When dealing with literal data within our application we know that the length of that data cannot change, but when we take input from any source outside of our application we have no control over

the availability or length of the data supplied. This will be most visible when taking input from the user. If we ask a user to enter their name with a maximum name size of 64 characters you can guarantee someone will attempt to input a name that is 500 characters long. Our buffer can only hold 64 so we now have 436 characters attempting to be written to “Somewhere” in memory. Let’s hope the extra characters are not overwriting an important OS dll in RAM.

The flip side is when a malicious actor can take advantage of this buffer overrun to inject a short piece of malicious code into a machine’s active memory, or even exploit the overrun to read data (such as security keys) from other applications currently open in memory.

Always check that any data that is obtained from outside of your application, and most especially from users, is validated to confirm that it is the correct type of data and that it can never exceed the length of the memory buffer required to hold the data. In most cases this can be done using a simple If Then true/False test.

Most buffer overruns relate to “Strings” and many programming environments and libraries will make a “String Safe” library available as an additional level of safety when working with strings.

The C Runtime (CRT) contains both safe and unsafe string handling funcions, but as a new coder it can be difficult to work out what is safe and what is not. The “Unsafe” functions just mean you as the programmer are responsible for adding rules and checks to make the function safe and this is the default for Linux systems. Safe string handling function for the coder to add additional information to make the use of the function safer, but nothing is perfect. Microsoft makes a “strsafe.h” available for the C Run Time on Windows which contanes a set of “Safer String” functions for Windows. “banned.h” is another legacy file from Microsoft which lists the banned or “derecated” unsafe string handling functions. Reading from this file as well as some reaserch about “Safe Strings” will assist in understanding the functions to be very careful with. “strsafe.h” can be found in the .\include directory of MinGw.

The legacy “banned.h” can be found here: https://github.com/x509cert/banned
And an example of an alternative Safe String library can be found here:
https://github.com/intel/safestringlib

Another area of coding security to pay attention to is “Data Security”. Any time that sensitive information or “Data” is moved from one location to another we need to assess the risk of that data being intercepted by a 3rd party either by accident or by malicious activity.

This will become most important when transporting data over any medium outside of your application routines such as over a local network, or via the internet. There are many different encryption and encoding schemes available to safeguard data during its transport.

User authentication may also be required in some coding applications such as server logins, or web account logins. These are sometimes complex in practice, but there are many good articles and libraries available to assist with further study.

Code security is a somewhat advanced subject that will require further study as you become more familiar with programming and coding. In the meantime introduce a habit of testing the length and type of data that you are importing into your application, most especially data that comes from user input. Don’t attempt to over-complicate code for security. Keeping your code simple makes it easier for yourself and others to read and identify security issues that you may have missed when first writing the code. Simple code will have a lower likelihood of introducing hidden security holes.

Below is a quick summary of basic code security elements.

- Input Validation
- Output Encoding
- Authentication and Password Management (includes secure handling of credentials by external services/scripts)
- Session Management
- Access Control
- Cryptographic Practices
- Error Handling and Logging
- Data Protection
- Communication Security
- System Configuration
- Database Security
- File Management
- Memory Management
- General Coding Practices

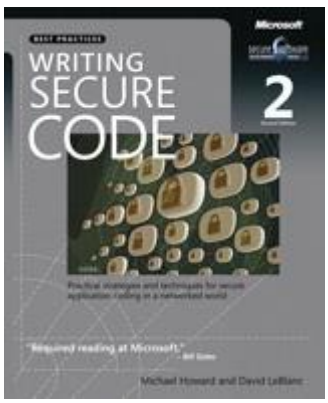
Further reading on code security and code vulnerabilities:

OWASP, Open Web Application Security Project

OWASP Secure Coding Practices-Quick Reference Guide

Download the current v2 (Stable) release PDF:

Writing Secure Code, 2nd Edition By David LeBlanc, Michael Howard



Conclusion

Having an understanding of computer technologies and programming is synonymous with having a knowledge of mathematics and language. It affords us the ability to understand the possibilities and make sound decisions in both our personal and professional lives. Even if we don't choose a career in ICT, understanding programming can benefit our lives in many ways.

The fundamentals of programming extend across the boundaries of the common programming languages and I hopefully have illustrated the differences and more importantly the similarities between different languages. Learning the fundamental principles of programming and applying them in even one of the common languages at the start will make it easier to understand and learn other languages as we progress.

Learning programming and coding does have its inherent challenges, but so too does learning to read and write and understand mathematics. If we can we can learn to read and write, understand essential mathematical principles then we can also learn to program and write code.

Appendix

[Understanding Characters, Unicode and the function of the Graphics Device Interface]

Hyphens and Dashes

The most common hyphen character is that which exists on a typewriter and remains as a legacy character "hyphen-minus" '-' on modern electronic keyboards. When typewriters and keyboards were first created there was a necessity to limit the amount of print charters due to space restrictions, so the hyphen-minus became the defacto of hyphen, en dash, em dash and minus sign all lumped together in one key.

In the real world of literacy there are many different variants of the hyphen and dash, each with its own meaning and intended use. The most common being '-' Hyphen-minus, '–' En Dash and '—'Em Dash.

In the computer world the most common hyphen is the minus sign '-' and has a value of:
Dec:45 Hex:2D Bin:00101101 HTML:-
The '-' hyphen-minus is the only hyphen/dash ever used in source code, except for what exists within a string variable.

Not all applications, including word processors, are capable of handling every type of character in existence, and although some characters may have changed or been removed from different standards, the Hexadecimal values may still exist in some documents.

Below is a table showing some of the hyphen and dash characters.	
	Recommended use.
' '	Denotes a glyph that does not display correctly in MS Word.

	UtF-8 / UTF-6 conversion issues or deprecated.
	May be deprecated.

Decimal order	Hex(ANSI)	Hex(UTF-8)	Hex(UTF-16)	HTML 4/5	Glyph	Description
45	2D	00 2D	00 2D	&#45	-	hyphen-minus
126	7E	00 7E	00 7E	~	~	tilde
173	AD	C2 AD	00 AD		‘-‘	soft hyphen
8208		E2 80 90	20 10	‐	-	hyphen
8209		E2 80 91	20 11	‑	-	non-breaking hyphen
8210		E2 80 92	20 12	&#8210	–	figure dash
8211		E2 80 93	20 13	– –	–	en dash
8212		E2 80 94	20 14	— —	—	em dash
8213		E2 80 95	20 15	―	—	horizontal bar
8275		E2 81 93	20 53	⁓	~	swung dash
8315		E2 81 BB	20 7B	⁻	⁻	superscript minus
8331		E2 82 8B	20 8B	₋	₋	subscript minus
8722		E2 88 92	22 12	− −	−	minus sign
11834		E2 B8 BA	2E 3A	⸺	‘——‘	two-em dash
11835		E2 B8 BB	2E 3B	⸻	‘———	three-em dash
65073		Ef E8 B1	FE 31	︱	｜	presentation form for vertical em dash
65074		Ef B8 B2	FE 32	︲	｜	presentation form for vertical en dash
65112		Ef B9 98	FE 58	﹘	”	small em dash
65123		Ef B9 A3	FE 63	﹣	”	small hyphen-minus
Uses decimal Integer		**Uses Hex value (Hexadecimal)**	**Uses Hex value (Hexadecimal)**	**Uses decimal value (Integer)**		

So Glyph:– = Dec:8211 = UTF-16-Hex:0x2014 = UTF-16-Hex:U+2014 = HTML5-Dec: – = UTF-8-Hex:0xE28093

U+2014 is just another way of writing 0x2014, some programming languages like C may use \u2014 or python u"\u2014" which all represent a UTF-16 character.

(NOTE! Python 3 uses UTF-8 by default for all strings).

Another common character error to look out for is what is known as a Byte Order Mark (BOM). BOMs are sometimes written as 2 bytes at the beginning of a file to denote its “endianness” and will appear as emojibake when opened in an editor using the incorrect encoding for the file. The following table shows in the right hand column the characters you may encounter at the beginning of a document.

Encoding	Representation (hexadecimal)	Representation (decimal)	Bytes as CP1252 characters
UTF-8[a]	EF BB BF	239 187 191	ï»¿
UTF-16 (BE)	FE FF	254 255	þÿ
UTF-16 (LE)	FF FE	255 254	ÿþ
UTF-32 (BE)	00 00 FE FF	0 0 254 255	^@^@þÿ (^@ is the null character)
UTF-32 (LE)	FF FE 00 00	255 254 0 0	ÿþ^@^@ (^@ is the null character)
UTF-7[a]	2B 2F 76[b][13][14]	43 47 118	+/v
UTF-1[a]	F7 64 4C	247 100 76	÷dL
UTF-EBCDIC[a]	DD 73 66 73	221 115 102 115	Ýsfs
SCSU[a]	0E FE FF[c]	14 254 255	^Nþÿ (^N is the "shift out" character)
BOCU-1[a]	FB EE 28	251 238 40	ûî(
GB-18030[a]	84 31 95 33	132 49 149 51	„1•3

Links:

https://en.wikipedia.org/wiki/Soft_hyphen

https://www.stylemanual.gov.au/grammar-punctuation-and-conventions/punctuation-and-capitalisation/dashes

https://www.merriam-webster.com/words-at-play/em-dash-en-dash-how-to-use

https://en.wikipedia.org/wiki/Dash

https://unicodeplus.com/category/Pd

https://alistapart.com/article/emen/

https://www.grammarly.com/blog/hyphens-and-dashes/

Software code editors

It is a common misconception that computers understand or can “See” what is displayed on a screen or monitor. This could not be further from the truth as computers have absolutely no understanding of what a picture is. In fact, your computer cannot even understand or read text that appears on your screen, nor can it see your mouse cursor. Everything is an image, everything you see on your screen, including text is nothing more than a drawn picture. The image being drawn to a monitor is the very last action that occurs in a complex chain of mathematical events, and no other actions occur at the monitor.

Your computer only understands binary numbers in blocks of “Bytes” from ‘0000’ to ‘1111’. Your computer is capable (internally) of translating Hexadecimal and decimal representations of these binary bytes as ASCII Dec and Hex, but this is really only visible to us at the compiler and code editor level. Everything at that level is considered to be an integer, even characters. A computer can only understand and manipulate representations of binary bytes. Groups of binary bytes can be used mathematically to represent more complex integers. These more complex groups of bytes are usually placed into categories called “Types”. A standard integer (int) “Type” on a 32 bit computer is 32 bits wide, in other words it is 4 bytes wide. On a 64 bit computer an integer (int) is 64 bits or 8 bytes wide.

Unicode characters are stored in “Types” according to their total size value for the largest character in binary bits. A UTF-8 character type is one byte wide. UTF-8 can combine up to 4 x UTF-8 types(4 x 1 byte) to contain its largest character value. UTF-16 is 2 bytes wide and can combine 2 x UTF-16 types to hold its largest character value.

The ASCII character set describes what can be manipulated by a computer as it represents a total of 1 byte. When coding we only use the first 7 bits of the byte or 128 (0 to 17) for character representation, and the last 8th bit is used for signed (negative) integers, error checks and for representing binary data. All characters from Dec 32 to Dec 126 are printable characters (aka what you can use on a typewriter and basic characters on your keyboard) that are usable when entering code into a code editor. You have 95 valid characters that the compiler understands and are legitimate characters for writing source code.

We can make use of the other characters in the list of Dec from 0 to 255 if required but we must enter them as a binary number or as a hexadecimal number as this is all that the compiler can correctly interpret. Usually we will use a Hex value in the form of 0x(nibble)(nibble), so to make use of a line return (Windows CRLF) we will need to use Hex ‘0x0A’,’0x0D’ which is 2 bytes. Many programming languages will also use what is known as escape characters to represent the hexadecimal value. CRLF is then shown as ‘\r’,’\n’.

ASCII Printer characters					
Dec	Hex	Binary	HTML	Char	Description
Start of “Control Characters”					
0	00	00000000	&#0;	NUL	Null
1	01	00000001	&#1;	SOH	Start of Header
2	02	00000010	&#2;	STX	Start of Text
3	03	00000011	&#3;	ETX	End of Text
4	04	00000100	&#4;	EOT	End of Transmission
5	05	00000101	&#5;	ENQ	Enquiry
6	06	00000110	&#6;	ACK	Acknowledge
7	07	00000111	&#7;	BEL	Bell
8	08	00001000	&#8;	BS	Backspace

9	09	00001001		HT	Horizontal Tab
10	0A	00001010
LF	Line Feed
11	0B	00001011	&#11;	VT	Vertical Tab
12	0C	00001100		FF	Form Feed
13	0D	00001101
CR	Carriage Return
14	0E	00001110	&#14;	SO	Shift Out
15	0F	00001111	&#15;	SI	Shift In
16	10	00010000	&#16;	DLE	Data Link Escape
17	11	00010001	&#17;	DC1	Device Control 1
18	12	00010010	&#18;	DC2	Device Control 2
19	13	00010011	&#19;	DC3	Device Control 3
20	14	00010100	&#20;	DC4	Device Control 4
21	15	00010101	&#21;	NAK	Negative Acknowledge
22	16	00010110	&#22;	SYN	Synchronize
23	17	00010111	&#23;	ETB	End of Transmission Block
24	18	00011000	&#24;	CAN	Cancel
25	19	00011001	&#25;	EM	End of Medium
26	1A	00011010	&#26;	SUB	Substitute
27	1B	00011011	&#27;	ESC	Escape
28	1C	00011100	&#28;	FS	File Separator
29	1D	00011101	&#29;	GS	Group Separator
30	1E	00011110	&#30;	RS	Record Separator
31	1F	00011111	&#31;	US	Unit Separator
Start of “Printable Characters”					
32	20	00100000		space	Space
33	21	00100001	!	!	exclamation mark
34	22	00100010	"	"	double quote
35	23	00100011	#	#	number
36	24	00100100	$	$	dollar
37	25	00100101	%	%	percent
38	26	00100110	&	&	ampersand
39	27	00100111	'	'	single quote
40	28	00101000	(	(	left parenthesis
41	29	00101001	)	)	right parenthesis
42	2A	00101010	*	*	asterisk
43	2B	00101011	+	+	plus
44	2C	00101100	,	,	comma
45	2D	00101101	-	-	minus
46	2E	00101110	.	.	period
47	2F	00101111	/	/	slash
48	30	00110000	0	0	zero
49	31	00110001	1	1	one
50	32	00110010	2	2	two
51	33	00110011	3	3	three
52	34	00110100	4	4	four
53	35	00110101	5	5	five
54	36	00110110	6	6	six
55	37	00110111	7	7	seven
56	38	00111000	8	8	eight
57	39	00111001	9	9	nine
58	3A	00111010	:	:	colon

59	3B	00111011	&#59;	;	semicolon
60	3C	00111100	&#60;	<	less than
61	3D	00111101	&#61;	=	equality sign
62	3E	00111110	&#62;	>	greater than
63	3F	00111111	&#63;	?	question mark
64	40	01000000	&#64;	@	at sign
65	41	01000001	&#65;	A	
66	42	01000010	&#66;	B	
67	43	01000011	&#67;	C	
68	44	01000100	&#68;	D	
69	45	01000101	&#69;	E	
70	46	01000110	&#70;	F	
71	47	01000111	&#71;	G	
72	48	01001000	&#72;	H	
73	49	01001001	&#73;	I	
74	4A	01001010	&#74;	J	
75	4B	01001011	&#75;	K	
76	4C	01001100	&#76;	L	
77	4D	01001101	&#77;	M	
78	4E	01001110	&#78;	N	
79	4F	01001111	&#79;	O	
80	50	01010000	&#80;	P	
81	51	01010001	&#81;	Q	
82	52	01010010	&#82;	R	
83	53	01010011	&#83;	S	
84	54	01010100	&#84;	T	
85	55	01010101	&#85;	U	
86	56	01010110	&#86;	V	
87	57	01010111	&#87;	W	
88	58	01011000	&#88;	X	
89	59	01011001	&#89;	Y	
90	5A	01011010	&#90;	Z	
91	5B	01011011	&#91;	[	left square bracket
92	5C	01011100	&#92;	\	backslash
93	5D	01011101	&#93;	]	right square bracket
94	5E	01011110	&#94;	^	caret / circumflex
95	5F	01011111	&#95;	_	underscore
96	60	01100000	&#96;	`	grave / accent
97	61	01100001	&#97;	a	
98	62	01100010	&#98;	b	
99	63	01100011	&#99;	c	
100	64	01100100	&#100;	d	
101	65	01100101	&#101;	e	
102	66	01100110	&#102;	f	
103	67	01100111	&#103;	g	
104	68	01101000	&#104;	h	
105	69	01101001	&#105;	i	
106	6A	01101010	&#106;	j	
107	6B	01101011	&#107;	k	
108	6C	01101100	&#108;	l	
109	6D	01101101	&#109;	m	

110	6E	01101110	&#110;	n	
111	6F	01101111	&#111;	o	
112	70	01110000	&#112	p	
113	71	01110001	&#113;	q	
114	72	01110010	&#114;	r	
115	73	01110011	&#115;	s	
116	74	01110100	&#116;	t	
117	75	01110101	&#117;	u	
118	76	01110110	&#118;	v	
119	77	01110111	&#119;	w	
120	78	01111000	&#120;	x	
121	79	01111001	&#121;	y	
122	7A	01111010	&#122;	z	
123	7B	01111011	&#123;	{	left curly bracket
124	7C	01111100	&#124;	|	vertical bar
125	7D	01111101	&#125;	}	right curly bracket
126	7E	01111110	&#126;	~	tilde
127 is also a "Control Character"					
127	7F	01111111	&#127;	DEL	delete
Start of "Binary Values" 128 To 255					

So how do we use Unicode characters?

Most modern code editors will be capable of understanding at a minimum ANSI (ASCII) but many will also be UTF-8 capable. The only time we use Unicode characters when coding is if they are passed to a variable in their binary (Hex) form or when writing string literals in our source code. To create a Unicode string literal in an ANSI editor you will need to create the entire string from Hex values Hello -> {0x48, 0x65, 0x6C, 0x6C 0x6F}. Some languages may use escape sequences instead of 0x such as \x6C. \b6C \h6C. This example is still using ASCII characters, so to use Unicode most editors will require that the string be identified as a Unicode string and prefixed with an identifier.

UTF-16 example -> MyString = U{0x0048, 0x0065, 0x006C, 0x006C 0x006F}

If your code editor is UTF-8 aware then you can enter the actual UTF-8 characters as the string literal, but it must still be identified as a Unicode string.

UTF-8 example -> MyString = U"Hello"
and an example using international characters (UTF-8).

MyString = U"Привет"
MyString = {0x041F, 0x0440, 0x0438, 0x0435, 0x0442}

Some Unicode-aware languages such as Python 3 may drop the u completely from strings as all strings default to UTF-8. Care must still be taken when accessing other encodings such as UTF-16.

Some languages may also allow you to use the Decimal order values[ambiguous] of the Unicode character. Decimal values are universal and unambiguous in relation to the glyph it represents, but do not state the Unicode encoding that is used (ANSI/UTF-8/UTF-16/etc.) The encoding type is left to whatever the moment to moment usage defaults to. The decimal list value or order of occurrence

from 0 to infinity [ambiguous get total know range of Unicode characters]] is not to be confused with the decimal value of an individual byte in a multibyte character…

Dec:8208|UTF-8-Hex:E2,80,90 UTF-8-Dec: 226,128,144| UTF-16-Hex:20,10, UTF-16-Dec:032,016

U+0000 Uses the hexadecimal values of UTF-16.

So, we write source using only the 95 ASCII characters, and can use Unicode only in string literals, often using an identifier, and write as a Hex value if it is an ANSI editor. We can pass Unicode data via variables as hexadecimal values.

[feff0000 <- UTF-8 Zero with BOM]

[Insert very brief explanation of memory, LE/BE and BOM]
[The dangers of copy/paste from RTF word processors and web pages]

Back to what I said about text not being displayed on the monitor screen…

Glyphs…

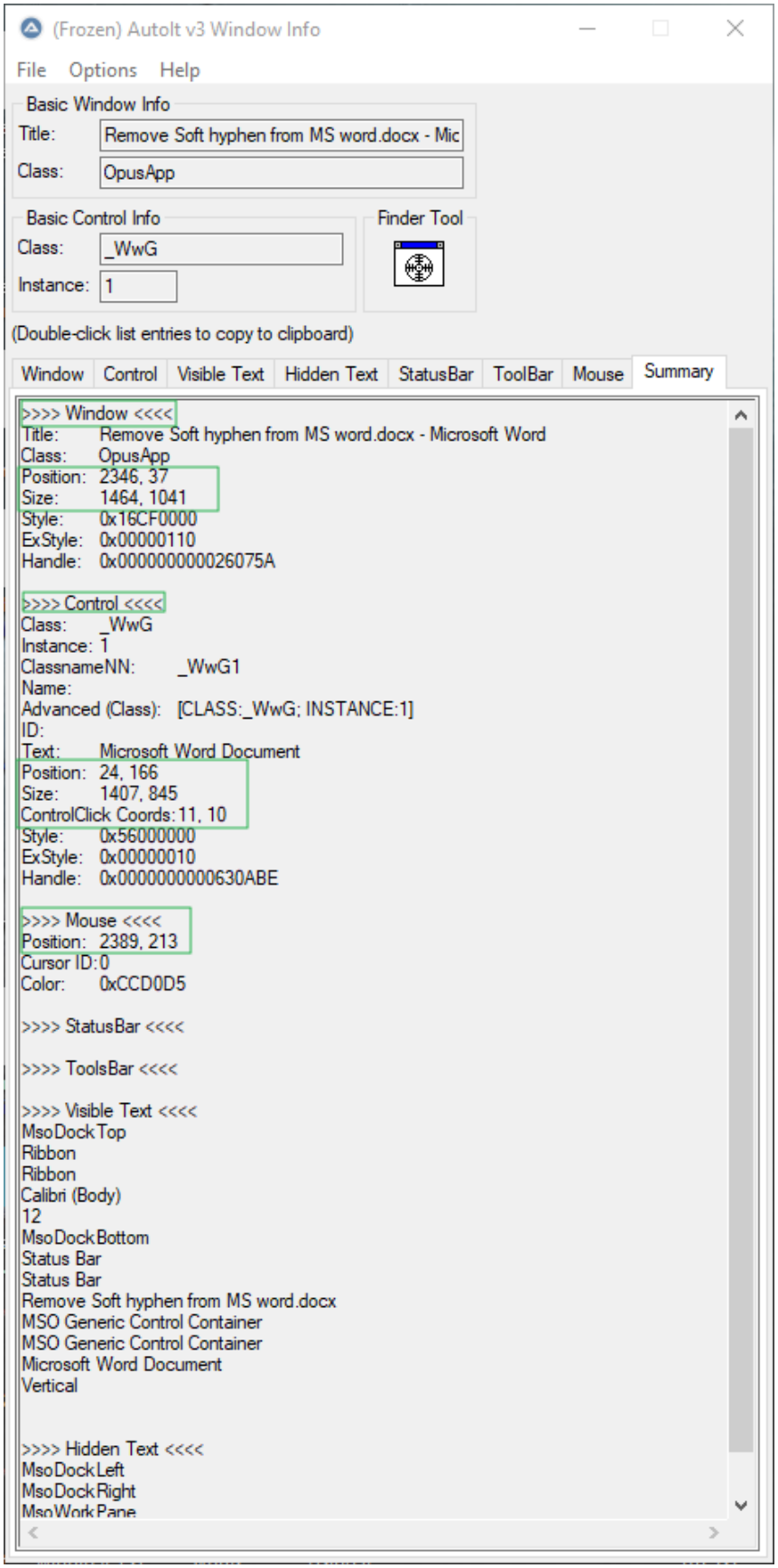
All of the information (Image) that you see on your monitor is stored as data structures which contain the ID, x,y positions as well as the z order of all windows and images that exist on the display. Everything happens here and this is where all of the curser, mouse movements and keyboard inputs are updated. When you move your mouse all that changes is the x,y data structure for the pointing device. The new x,y position as well as the image for the pointer is sent to the Graphics Display Interface (GDI) and a new collage of layered images is assembled and ultimately flattened to a single image and sent to the monitor as an update.

The same occurs with all text that you can see on the screen as it is all just images pasted according to their x,y positions in relation to the windows in which they are pasted. A character is just a Hexadecimal (Binary) number that relates to a character. That number selects a particular character, it then selects the font type and size and lastly copies the bitmap image of that font to the correct x,y coordinates in which it is being used, assembled with all other images in the GDI and sent to the monitor as a single image.

The same when selecting text, the mouse ‘x,y start left button down’ is recorded and ‘end x,y position of left button up’ is recorded. The 2 coordinates are compared against the data structures of the current windows in the z order. When the appropriate windows’ data structure is found, it then finds the array of integers that match the mouse x,y start and end and repeats the update process of the current character bitmap, this time with an extra image of transparent blue in a new layer over the top of the characters. Ultimately the GDI and screen are updated every time there is a change in the data structures.

It’s important to forget about any concept of “Things” happening “on” your monitor in the world of coding. All of the magic occurs in a collection of data structures organised by the OS and the GDI. The screen image update is the last thing that occurs.

The following is a quick summary of the data structures used in the canvas area of this document. In it you can see the relative window and mouse coordinates.


(Frozen) AutoIt v3 Window Info
File Options Help
Basic Window Info
Title: Remove Soft hyphen from MS word.docx - Mic
Class: OpusApp
Basic Control Info
Class: _WwG
Instance: 1
Finder Tool
(Double-click list entries to copy to clipboard)
Window Control Visible Text Hidden Text StatusBar ToolBar Mouse Summary
>>>> Window <<<<
Title: Remove Soft hyphen from MS word.docx - Microsoft Word
Class: OpusApp
Position: 2346, 37
Size: 1464, 1041
Style: 0x16CF0000
ExStyle: 0x00000110
Handle: 0x000000000026075A
>>>> Control <<<<
Class: _WwG
Instance: 1
ClassnameNN: _WwG1
Name:
Advanced (Class): [CLASS:_WwG; INSTANCE:1]
ID:
Text: Microsoft Word Document
Position: 24, 166
Size: 1407, 845
ControlClick Coords: 11, 10
Style: 0x56000000
ExStyle: 0x00000010
Handle: 0x0000000000630ABE
>>>> Mouse <<<<
Position: 2389, 213
Cursor ID: 0
Color: 0xCCD0D5
>>>> StatusBar <<<<
>>>> ToolsBar <<<<
>>>> Visible Text <<<<
MsoDockTop
Ribbon
Ribbon
Calibri (Body)
12
MsoDockBottom
Status Bar
Status Bar
Remove Soft hyphen from MS word.docx
MSO Generic Control Container
MSO Generic Control Container
Microsoft Word Document
Vertical
>>>> Hidden Text <<<<
MsoDockLeft
MsoDockRight
MsoWorkPane

References

[Add references, useful web links…]