**February 2023**



**Crown Programming Language**

**Reference Manual**

Version 1.0 – Humble Pig

Developed by Gabriel Margarido

**Resume**

Crown programming language compiler aims to simplify C programming language syntax and built-in features. Such as: string handling, vectors handling, file handling, parser, tokenization and compiler development. Normally we think C programming is hard and complex, this is actually truth. However with Crown programming language, the scenery has been changed, with a syntax near to Ruby, Javascript or Lua, even more people can program in C without programming in C. Using an intermediate C-transpiled language, that supports all C native functions and libraries, it means, you can program for Arduino, C Microcontrollers, desktop and even mobile computers.

Memory isn’t automatic disallocated or manipulated, Crown programming language runs directly on binary code, without a runtime or memory wasting due to a runtime or virtual machine. It’s syntax is easier than C, C++ or even Rust.

Some syntax elements are inherited from Pascal, Go, Ruby, Lua, Javascript and C.

All low-level functionalities are inherited from C programming language.

It’s also a procedural, weak and static typed compiled programming language. Crown does not support classes or inheritance, due to it’s not an object-oriented programming language, like C++ (for instance).

Crown programming language compiles on pure ANSI-C, it means that all computers that has a Standard ANSI-C compiler can run Crown, independently of the operating system or processor architecture.

**Gabriel Margarido,**

**February 2023**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Compiling and installing Crown compiler and TinyCC from sources:**

1. Install these softwares first: GNU Make, Node.js 12+, NPM 8+, GCC

and Clang/LLVM. (And also **Git Bash** if you are running in Windows.)

On Ubuntu or Debian you can run:

sudo apt install clang gcc nodejs npm make

2. (unzip and enter inside the downloaded directory,

next run the following commands)

sudo make all install

They’re gonna be installed at:

/usr/local/bin/crown

/usr/local/bin/tcc

4. To compile a source file

crown <file>.crown

You can uninstall Crown Compiler and TinyCC by running:

sudo make remove

Visual Studio Code Extension available inside **vscode** directory,

just copy **crown-syntaxhighlight**  to:

**MacOS/Linux: ~/.vscode/extensions/**

**Windows: %USERPROFILE%\.vscode\extensions**

**You can also run the following command from vscode directory to install Crown VSCode Extension:**

MacOS/Linux: **cp -Rfv crown-syntaxhighlight ~/.vscode/extensions/**

Windows:  **xcopy crown-syntaxhighlight %USERPROFILE%\.vscode\extensions\crown /E /H /C /I**

These are all existing datatypes in Crown Programming Language:

|  |  |  |
| --- | --- | --- |
| **Datatype** | **Numeric Value** | **Description** |
| **int** | -32.768 to 32.767 | Integer value |
| **float** | 3,4E-38 to 3.4E+38 | Decimal positive or negative real value |
| **String** | -128 to 127 | Immutable/Standard string |
| **\*String** | -128 to 127 | Mutable string |
| **bool** | 0 or 1 | true or false / 0 or 1 |
| **ulong\_int** | 0 to 4.294.967.295 | Big integer value |
| **long\_int** | -2.147.483.648 to 2.147.483.647 | Big positive or negative integer value |
| **long\_float** | 3,4E-4932 to 3,4E+4932 | Big decimal real value |
| **void** | None | Empty value (most used in function declaration) |
| **mathematical** | None | Mathematical expressions |
| **Macro** | Macro/Constant:  Any datatype. | Create the most optimized constant on memory |

Variable declaration:

int a = 13

float b = 457.89

float bc = -458.76

\*String name = “Maria Juliana”

String surname = “Gomez”

String other\_name[64] = “Mariana Julia Andressa Conda”

bool c = true

bool e = false

ulong\_int my\_number = 409567328

long\_int my\_negative\_number = -45

long\_int my\_big\_negative\_number = -458670382

long\_float my\_big\_negative\_number = -410492230.89045

mathematical my\_expression = “(2\*(34.5+78-12.9)+45)/4”

Macro name = “Gabriel Margarido”

Macro age = 17

Macro salary = 137.56

Macro isOk = true

Macro isOk = false

print(“%s”, name)

print(“%i”, age)

print(“%f”, salary)

float a = 456.70

int b = 4

float c = 2.5

mathematical my\_expression = “(2\*(a\*a+b\*2)+45)/c\*2”

Variable reassignment:

int a = 32

a := 4

float b = 28.5

b := 3.45

\*String c = “Hello world”

c := “Bye world”

bool f = true

f := false

Arrays declaration with automatic size definition:

int[] a = (0, 2, 4, 6, 8, 10, 12, 14)

float[] b = (0.58, 2.47, 6.78, 8.23, 10.50, 12.38)

String[] c = (“Hello world”, “Bye world”, “See you later”)

**Note:** String[] and \*String[] arrays are the same thing. Do not confound with String[] and \*String[] variables!

Arrays declaration with manual size definition:

int[] a[8] = (0, 2, 4, 6, 8, 10, 12, 14)

float[] b[6] = (0.58, 2.47, 6.78, 8.23, 10.50, 12.38)

String[] c[3] = (“Hello world”, “Bye world”, “See you later”)

**Note:** String[] and \*String[] arrays are the same thing. Do not confound with String[] and \*String[] variables!

Every time an array is declared and initialized,

an integer variable named <array\_name>\_size

containing the length of the array is automatically created.

In this case, two variables were created: a\_size and b\_size

We can see these length integer values, by printing them on the screen.

print(“The A size is: %i \n”, a\_size)

print(“The B size is: %i \n”, b\_size)

Showing messages on the screen:

print(“Hello world\n”)

\*String msg

print(“%s”, msg)

Formatted print on the screen:

|  |  |
| --- | --- |
| **Reference** | **Value** |
| %i | int |
| %f | float | mathematical |
| %s | String | \*String |
| %c | String |
| %d | int | decimal |

print(“The selected sumber was: %i”, my\_number)

print(“My salary is: %f”, my\_salary)

print(“My name is: %s”, my\_name)

Reading data from user:

input(“%s”, name)

print(“Your name is: %s”, name)

Showing arrays on the screen:

dump int -> a

dump float -> b

Removing last element of the array:

drop -> a

drop -> b

Shifting element in the array:

Change value of index [3] in array **b** to 145

shift b -> 145 in 3

Write string to text file:

File myfile = openfile(“testing.txt”, “w”)

io.write(myfile,”%s”,”Hello world from text\n”)

closefile(myfile)

Read from text file:

io.read “testing.txt” -> mybuffer

print(“%s”, mybuffer)

Calculate factorial from an integer number:

int factorial = fat(20)

print(“%i”, factorial)

If-Conditional

if (a > 3) do

print(“First condition”)

elseif (b <= 4) do

print(“Second condition”)

else

print(“None of above”)

end

When-Conditional

while (true) do

when (a > 3) do

print(“First condition”)

break

elsewhen (b <= 4) do

print(“Second condition”)

break

else

print(“None of above”)

break

end

end

Repetition loops

for i in 0..5 do

print(“Here am I”)

end

5 times do

print(“Here am I”)

end

while (a > 3) do

print(“Here am I”)

end

Human-readable operators: is isnot and or  
Machine-readable operators: == != && ||

while (a is 3) do

print(“Here am I”)

end

int a = 4

if (a is 4) do

...

elseif (a isnot 5)

...

end

int b = 7

while (a is 4 and b is 7) do

...

end

Writing mathematical expressions on the screen

float a = 456.70

int b = 4

float c = 2.5

mathematical my\_expression = “(2\*(a\*a+b\*2)+45)/c\*2”

print(“%f”, my\_expression)

Function declaration:

**Function without return**

fn my\_function() -> void do

...

end

**Function with integer return**

fn my\_function() -> int do

...

return 3

end

**Function with float/real return**

fn my\_function() -> float do

...

return 3.67

end

**Function with boolean return**

fn my\_function() -> bool do

...

return true

end

**Function with (mandatory) mutable string return**

fn my\_function() -> \*String do

...

return “Hello world”

end

fn my\_function() -> String do

...

return “Hello world”

end

Function arguments:

**Immutable string argument**

fn my\_function(char name[], char surname[]) -> void do

...

print(“You are: %s %s\n”, name, surname)

end

**Mutable string argument**

fn my\_function(String name, String surname) -> void do

...

print(“You are: %s %s\n”, name, surname)

end

fn my\_function(\*String name, \*String surname) -> void do

...

print(“You are: %s %s\n”, name, surname)

end

**String slicing**

String name = “Jean Brawicz”

String.slice(name,” ”)

print(“%s”, name)

**Non-initialized string declaration: 32 characters of length**

String reg[32] = null

**Extracting substring from string**

String msg = “Hello world”

String substring[128] = null

String.substring msg from 0 to 3 -> substring

**Concatenating strings**

String msg\_first = “Hello ”

String msg\_second = “World”

String.concat(msg\_first, msg\_second)

print(“%s \n”, msg\_first)

**Getting length of string**

String msg\_first = “Hello World”

int length = String.len(msg\_first)

print(“%i \n”, msg\_length)

**Putting string to lowercase**

String msg = “HELLO WORLD”

String lower = String.lowercase(msg)

print(“%s \n”, lower)

**Putting string to uppercase**

String msg = “hello world”

String upper = String.uppercase(msg)

print(“%s \n”, upper)

* **Selecting features on source-code compilation**

1. Start new program overwriting old program (mandatory for correctly working!)

using crown

2A. Use GCC (GNU C Compiler) to compile automatically written source-code.

using gcc

2B. Use Clang (Clang/LLVM) to compile automatically written source-code.

using clang

2C. Use TinyCC (Tiny C Compiler) to binary compile automatically written source-code.

using tinycc

2D. Use TinyCC VM (Tiny C Virtual Machine) to bytecode compile automatically written source-code.

using tinycc\_vm

3. See automatically generated C code from source-code.

using debugging

**Main program structure (with GCC)**

using crown

using gcc

fn main() -> int do

# ”Your program goes here”

...

return 0

end

**Main program structure (with GCC)**

using crown

using gcc

fn main() -> int do

# ”Your program goes here”

...

return 0

end

**Main program structure (with Clang)**

using crown

using clang

fn main() -> int do

# ”Your program goes here”

...

return 0

end

**Main program structure (with TinyCC Compiler)**

using crown

using tinycc

fn main() -> int do

# ”Your program goes here”

...

return 0

end

**Main program structure (with TinyCC Virtual Machine)**

using crown

using tinycc\_vm

fn main() -> int do

# ”Your program goes here”

...

return 0

end

**Import additional native library**

Import C native compiler libraries, all libraries below

are automatically imported when program starts.

import “stdio.h”

import “stdlib.h”

import “string.h”

import “ctype.h”

**Import custom external modules**

Import C external module libraries, such as

Lua Programming Language Modules.

import\_module “lua.h”

**Call C native functions**

Call native C functions without changing anything.

system(“ls”)

system(“pause”)

system(“free -h”)

**Create structures**

Create pseudo-classes, then you can

create pseudo-objects from them.

A. Here we are creating the structure

struct **Vehicle** do

float weight = null

int year = null

String model[32] = null

bool isRunning = null

endstruct

B. Now, creating the pseudo-objects

Vehicle **toyota** = null

Vehicle **mitsubishi** = null

C1. Next, acessing Toyota pseudo-object

**toyota**.weight = 1.2

**toyota**.year = 2022

**toyota**.model = “Etios”

**toyota**.isRunning = true

C2. Next, acessing Mitsubishi pseudo-object

**mitsubishi**.weight = 2.6

**mitsubishi**.year = 2012

**mitsubishi**.model = “Pajero Sport”

**mitsubishi**.isRunning = false

D. Printing on the screen all properties of pseudo-objects

print(“%f\n”, **mitsubishi**.weight)

print(“%i\n”, **mitsubishi**.year)

print(“%s\n”, **mitsubishi**.model)

print(“%f\n”, **toyota**.weight)

print(“%i\n”, **toyota**.year)

print(“%s\n”, **toyota**.model)

**Acessing command line arguments (CLI args)**

The quantity of passed CLI arguments is stored inside an integer (int) argument counter, called: **argc**

And the passed arguments are stored inside an array of strings, called: **argv**

The first argument **argv[0]** stores the name of the called executable.

The position **argv[1]** stores the first argument.

The position **argv[2]** stores the second argument.

The position **argv[3]** stores the third argument.

And so on...

A. Acessing arguments inside **argv**

print(“Name of the executable: %s\n”, **argv[0]**)

print(“First argument: %s\n”, **argv[1]**)

print(“Second argument: %s\n”, **argv[2]**)

print(“Third argument: %s\n”, **argv[3]**)

...

**THE END**

**THIS SECTION IS STILL UNDER DEVELOPMENT!**

*WARNING! This is still in development and unstable, do not use this section! Severe bugs may occur…*

**Developing compilers with CDK – Compiler Development Kit**

*WARNING! Still in development, bugs may occur…*

By using CDK, we are able to develop compilers with a built-in development kit.

Here we using **tokenizer** feature with all standard features.

using crown

using tinycc

using debugging

using tokenizer

Then, we’re declaring the reserved function \_\_tokenize.

o*utfile* and *infile* are the input-file variable and the output-file variable.

fn \_\_tokenize() -> void do

\_\_initialize with outfile and infile do

@compiler

equals stack[i] and “my\_special\_token” do

\*String next\_token = stack[i+1}

io.write(outfile, “%s”, next\_token)

end

equals stack[i] and “my\_other\_token” do

\*String next\_token = stack[i+1}

io.write(outfile, “%s”, next\_token)

end

equals stack[i] and “my\_normal\_token” do

\*String next\_token = stack[i+1}

io.write(outfile, “%s”, next\_token)

end

end

free

end

Now we should call \_\_tokenize function inside main function

fn main() -> int do

io.read argv[1] -> source\_file

\_\_tokenize(source\_file)

return 0

end

The equals x and y do instruction is the same thing as do if x == y do, however equals is for comparing strings, and if is for comparing expressions. It’s semantical meaning differs from if instruction. We’re basically comparing if variable or string x is equal to variable or string y, if it’s true do the following condition. Normally, it’s writing the correspondent instruction in the target language to a text file (object code/low-level code).

Here the free instruction frees the allocated memory due to compiler input and output C-I/O. stack[i] variable is related to the current token/symbol, broken by spaces, unless double-quote strings, parenthesis expressions and brackets expressions. So you can see the following sequence of tokens:

**Analyzing rushed expressions: main focus: (first token)**

Here is the following instruction

stack[i] stack[i+1] stack[i+2] stack[i+3] stack[i+4]

This text is “Hello world for all”

stack[i] stack[i+1] stack[i+2] stack[i+3]

if (a > 3 and b < 2 ) do

stack[i] stack[i+1] stack[i+2]

int b = 4

stack[i] stack[i+1] stack[i+2] stack[i+3]

**Analyzing variables: main focus: “=”**

int b = 4

stack[i-2] stack[i-1] stack[i] stack[i+1]

<type> <name> = <value>

stack[i-2] stack[i-1] stack[i] stack[i+1]

**Analyzing arrays: main focus: “=”**

int[] b = [0, 2, 4, ...]

stack[i-2] stack[i-1] stack[i] stack[i+1]

<type> <name> = <value>

stack[i-2] stack[i-1] stack[i] stack[i+1]