# Documentation of Corn

0. Understanding corn commands and project

Once you have installed Corn, you can open a cmd and type in “corntest” to check if corn is installed correctly.

0.1. Creating a new Corn project and compiling

Shift+click in the folder you want your project created and click “Open command prompt here”.

Corn has only 2 commands!

- **corn create MyProject**

- **corn main.corn MyProjectAsExecutable**

0.1.1. Create a project

After you type in ‘corn create ProjectName’, a new folder called ProjectName will be created.

This folder will contain a ‘main.corn’ file.

This is where you will write the corn source code (or at least the start:/finish block)

There will be 2 other folders: ‘Binaries’ and ‘C++ Source Code’

‘C++ Source Code’ folder will contain the generated C++ code

‘Binaries’ folder will contain the final program executable.

0.1.2. Compile a project

Nagivate with the command prompt into the newly created ProjectName folder (or open a new one there).

Type in ‘corn main.corn ExecutableName”.

This will compile all of your corn code into a runnable .exe file, called ExecutableName.

Note: The background process is actually: Corn Code -> C++ Code -> .exe

Hint: You can add a ‘-r’ argument at the end to also run the program once it is compiled!

ex: corn main.corn Exec -r

Bug: If you compile it again under the same executable name, it will not be overwritten!

ex: corn main.corn Exe

corn main.corn Exe // this will not compile

If you want to recompile the program, make sure you delete the executable (or change the compilation executable name the next time you compile)

1. Basic Hello World program:

**start:**

print “Hello World”

**finish**

Note: There is no ‘;’ put at the end of instructions

2. Data types:

Variables are separated into 2 categories: primitives and Objects

By convention, all primitive types start with lower case letter and all object types start with capital letter (Object o)

There are 5 primitives: int, float, char, bool, string

2.1. Primitives

int i **=** 1337

float f **=** 13.422

char c **=** ‘c’ //note: ‘ is used for char

bool b **=** true

string s **=** “I am a string” //note: “ is used for string

Note: there is no such thing as double, short, long, etc.

Note: there is no such thing as a pointer, especially for primitives.

Primitives are always passed by value.

int a **=** 123

int b **=** 77

a **=** b

b **=** 420

print a // a is 77

print b // b is 420

2.1.1. int arithmetics (follow C/C++ standard)

int supports the following operators: +, -, \*, /, %, ++, --

All of these operators function as expected

int a **=** 4

int b **=** (a \* 3 - 1) / 3 // b is 3

b++ // b is 4

Note: all ‘would-be’ decimals are discarded when making int calculations

Note: numbers specified directly by value such as 353 or -24 are treated as ints.

3/2 = 1

All other primitives can be converted to int by using the toInt function

string s **=** “1337”

int a **=** toInt(s) // a is 1337

int b **=** toInt(13.37) // b is 13

int c **=** toInt(‘c’) // c is 99 // or use charCode(char)

Bug: to convert from bool to int use boolToInt(bool) instead

2.1.2. float arithmetics (follow C/C++ standard)

float uses the same operators as int

float f **=** 10.0 // f is 10.0

float g **=** 11 // g is 11.0

int i **=** 10 // i is 10, 10 IS NOT 10.0

float h **=** toFloat(i)

2.1.3. char arithmetics (follow C/C++ standard)

A char represents a single character, such as ‘1’, ‘k’, ‘R’, ‘@’, etc

The character for new line is ‘\n’, for tab ‘\t’, etc

2.1.4. bool arithmetics

There aren’t many… but you can use

toBool(“true”), toBool(1), toBool(‘t’)

2.1.4. string arithmetics

string supports the following operations:

string a **=** “hello”

string b **=** “world”

string c **=** a + b // c is “helloworld”

To access positions of a string, use the [ and ] operators

string a **=** “hi”

a[2] **=** ‘o’ // a is “ho”

Note: strings start at 1, NOT 0!! There is no such thing as a[0].

Some useful string functions are:

**string toString(int or float or char or bool or Object)**

**int length(string)** // returns the length in characters of a string

**bool contains(string bigString, char or string)**

//returns true if bigString contains the second argument

contains(“hello world”, “wor”) // this will return true

**int positionOf(string bigString, char or string)**

//returns the position index of the second argument

positionOf(“hello”, “llo”) // this will return 3

positionOf(“hello”, “arrr”) // this will return 0 (false)

**Array of <string> split(string, string delimiters)**

// splits the string into an Array of strings separated

// by any character given as the second argument

Array of <string> a **=** split(“Hi,I am,text”, “,”) //a[1] **=** “Hi” a[2] **=** “I am” a[3] **=** “text”

**string substring(string bigString, int startPosition, int length)**

substring(“abcdefgh”, 2, 4) // returns “bcde”

**string substringUntil or substringUntilPosition(string bigString, int position)**

substringUntil(“abcdef”, 3) // returns “abc”

**string substringFrom or substringFromPosition(string bigString, int position)**

substringFrom(“abcdef”, 3) // returns “cdef”

2.2. Flow control blocks

By flow control blocks we mean: if/else, while, for

if a **==** 20

print “Yes, a is 20”;

else if a **==** 30

print “A is 30”;

else

print “:(“;

To and a flow control block, add ‘;’ (and optionally use ‘:’ to open it)

while b **==** true:

loop(b)

print b

doSomething();

You do not have to add ‘(‘ and ‘)’ around the condition, but you can if you want

for int x **=** 1:30

print x;

2.3. Functions

To define a function, use the following syntax:

returnType functionName(arg1type arg1, arg2type arg2, etc):

//do something

end

Example:

int lengthOfString(string s):

if s **==** “”

return 0;

else

return length(s);

end

Note: This is row sensitive! You can’t write:

if s **==** “” return 0; // THIS IS INCORRECT

// You should dedicate the whole line to a flow control check

Note: Also, opening a flow control block on one line REQUIRES a ‘;’

Note: ‘else if’ is treated as a single flow control keyword! It requires only one ‘;’

2.4. Classes and Objects

Use the following syntax when creating a class:

class MyClass:

int data

void doMethod(int x):

// method block

end

end

All methods and attributes of a class are public by default, but you can use ‘public’ and ‘private’ modifiers

class MyClass:

private int data

public void doMethod(int x):

// method block

end

end

Inheritance is done using the keyword “extends”

class MyClass extends Object

Note: All classes extend a default class, Object.

Note: All classes automatically have a getType() method, which returns the class name

(ex: MyClass mc **=** new MyClass() \ print mc.getType() //prints “MyClass”

Note: All classes have a asString() method, which should be overriden

Note: All methods are automatically overriden by the subclass (equivalent of ‘virtual’)

Note: Classes CAN extend more than 1 class, but this will be changed in the future.

(ex: class MyClass extends Object, File:)

There is no such thing as a pointer to an object, or ‘->’.

All methods of an object are called using ‘.’ ( ex: mc.asString() )

Static members are defined using the word ‘static’

public static void doSomething():

// code

end

To call a static method or use a static property, use ‘.’ ( ex: Class.staticMethod() )

2.4.1. Operator overload

There is only one operator pair that can be overloaded: ‘[‘ and ‘]’

(You can not overload +, = and so on)

Overloading [] will result in returning something upon using yourObject[30]

ex: string s **=** “hello”

print s[3] // ‘3’

The [] operator for strings returns a char by default.

To overload the [] operator for an object, you must create a global function:

YourReturnedType elem(YourOverloadVariable var, int index):

return YourReturnedVariable

Ex: for string, it is by default as follows:

char elem(string var, int index):

@C++

return var[index + 1];

@Corn

end

// var[index]

This function MUST be called ‘elem’ and must take 2 parameters of any type

Example: You could write your own HashMap and have it as:

template<T> T elem(HashMap<T> hashMap, string key):

return hashMap.getElement(key);

HashMap<int> hashMap **=** new HashMap()

print hashMap[“mykey”]

2.4.2. Templates

You can define a template class or function by adding template<SomeClass> before the definition.

3. Source code files

You can easily split your code into multiple source code files, if they all have the .corn extension.

import “D:\\MyProject\\my\_library.corn”

import “my\_library.corn” // if it’s in the same folder as main.corn

This literally copy-pastes the code from that file into your new file.

Careful: If you have any import loops, such as FileA imports FileB and FileB imports FileA, it will crash!

You can import any kind of text. You can even import the inside of a function!

Example:

void myFunction():

import “E:\\MyFunctionBodies\\functionbody.corn”

end // Note: please do not do it like this, it is a very sensitive feature :s

4. Some useful default classes and functions (try them and see what happens!):

4.1. File

File.writeFile(“hello world”, “myFile.txt”)

string s **=** File.readFile(“myFile.txt”)  
4.2. Array

Array<Object> a **=** new Array<Object>(20)

print a[10]

print a // print a.asString()

print a.length

a.push(new Object(20))

Object o **=** a.pop()

a.shuffle()

4.3. Matrix

Matrix<Object> m **=** new Matrix<Object>(10, 10)

print m[5][5]

print m[5] // print m[5].asString()

m[5].shuffle() // A Matrix is actually a fancier Array<Array<T>>!

print m

print m.nRows

print m.nCols

m.addColumn()

m.addRow()

4.4. Global functions

wait(1000)

pause()

int x

read x // from the console

x **=** randomInt(1, 10)

float f **=** randomFloat(0, 1)

That’s all.

Have fun!

Chances are you’ll encounter lots of bugs.

Upcoming features: macros, sockets, threads