C·Wayv Manual

V·Liance Foundation

July 26, 2018

Todo list

Contents

Chapter 1

Introduction

1.1 What is C·Wayv?

C·Wayv consists of a high-level programming language, inspired of Action Script & Haxe like language, usually symbolized with a "Wave arrow" (C↝) as title or simply with a tilde (C~)

Based on C++, acting like an overlay, it's possible to mix C~/C++. You can also interact with GLSL and Javascript, in a single file, variables from different languages are inter-accessible.

C~ is strongly typed to have robust code and best performances. Main goal is to easily have a single code-base which compiles to multiple targets.

It use the powerful Cwc intelligent compiler, which achieve outstanding performances using highly optimized C++ compiler in backend.
Also this provide the ability to generate any platform output binary since a multitude of backend toolchain can be selected.
Currently, there is the available toolchains :

Toolchain	From	Target
LibRT	Windows	Windows, Linux
libRT(Debian)	Linux	Linux, Windows

Haxe abstracts away many target differences, but sometimes it is important to interact with a target directly, which is the subject of ?? (??).

1.2 Getting Started

The following program in C~ prints "Hello World" after being compiled and run:

```
1 class Main {
2   public function Main():Void {
3     Debug.fTrace("Hello World");
4   }
5 }
```

This can be easily tested by saving the above code to a file named `Main.cw` and invoking the Cwc Compiler

1.2.1 Required tools

Compilation process is greatly simplified, Cwc is the unique tool you need to build your code.

Grab the last version of Cwc compiler

Unzip archive, and run Cwc

1.2.2 First use of Cwc

First Cwc ask for:
- Demos: Feel free to learn from example.
See download section

- Setting the Cwc path to your environement. This is usefull when you need to access to Cwc from anywhere in Command-line

Part I

Language Reference

Chapter 2

Types

The C~ type system knows seven type groups:

Group	Type	Prefix
Numeric	Int, Float	n
String	String	s
Array	Array, QueueArray, Map	a
Object	Class Object	o
Function	Function	f
Delegate	Delegate	d
Postprocessing	Typedef	t

2.1 Basic Types

Basic types are `Bool`, `Float` and `Int`. They can easily be identified in the syntax by values such as

Arithmetic				
Operator	Operation	Operand 1	Operand 2	Return
`++`	increment	`Int`	N/A	`Int`
		`Float`	N/A	`Float`
`--`	decrement	`Int`	N/A	`Int`
		`Float`	N/A	`Float`
`+`	addition	`Float`	`Float`	`Float`
		`Float`	`Int`	`Float`
		`Int`	`Float`	`Float`
		`Int`	`Int`	`Int`
`-`	subtraction	`Float`	`Float`	`Float`
		`Float`	`Int`	`Float`
		`Int`	`Float`	`Float`
		`Int`	`Int`	`Int`
`*`	multiplication	`Float`	`Float`	`Float`
		`Float`	`Int`	`Float`
		`Int`	`Float`	`Float`
		`Int`	`Int`	`Int`
`/`	division	`Float`	`Float`	`Float`
		`Float`	`Int`	`Float`
		`Int`	`Float`	`Float`
		`Int`	`Int`	`Float`
`%`	modulo	`Float`	`Float`	`Float`
		`Float`	`Int`	`Float`
		`Int`	`Float`	`Float`
		`Int`	`Int`	`Int`
Comparison				
Operator	Operation	Operand 1	Operand 2	Return
`==`	equal	`Float/Int`	`Float/Int`	`Bool`
`!=`	not equal	`Float/Int`	`Float/Int`	`Bool`
`<`	less than	`Float/Int`	`Float/Int`	`Bool`
`<=`	less than or equal	`Float/Int`	`Float/Int`	`Bool`
`>`	greater than	`Float/Int`	`Float/Int`	`Bool`
`>=`	great than or equal	`Float/Int`	`Float/Int`	`Bool`
Bitwise				
Operator	Operation	Operand 1	Operand 2	Return
`~`	bitwise negation	`Int`	N/A	`Int`
`&`	bitwise and	`Int`	`Int`	`Int`
`|`	bitwise or	`Int`	`Int`	`Int`
`^`	bitwise xor	`Int`	`Int`	`Int`
`<<`	shift left	`Int`	`Int`	`Int`
`>>`	shift right	`Int`	`Int`	`Int`
`>>>`	unsigned shift right	`Int`	`Int`	`Int`

Equality For enums:

Enum without parameters Are always represent the same value, so `MyEnum.A == MyEnum.A`.

Enum with parameters Can be compared with `a.equals(b)` (which is a short for `Type.enumEquals()`).

Dynamic: Comparison involving at least one Dynamic value is unspecifed and platform-specific.

2.1.1 Bool

> Type: Bool
> Represents a value which can be either true or false.

Values of type `Bool` are a common occurrence in conditions such as `if` (??) and `while` (??). The following operators accept and return `Bool` values:

- `&&` (and)
- `||` (or)
- `!` (not)

Haxe guarantees that compound boolean expressions are evaluated from left to right and only as far as necessary at run-time. For instance, an expression like `A && B` will evaluate `A` first and evaluate `B` only if the evaluation of `A` yielded `true`. Likewise, the expressions `A || B` will not evaluate `B` if the evaluation of `A` yielded `true`, because the value of `B` is irrelevant in that case. This is important in cases such as this:

```
1 if (object != null && object.field == 1) { }
```

Accessing `object.field` if `object` is `null` would lead to a run-time error, but the check for `object != null` guards against it.

2.1.2 Void

> Type: Void
> Void denotes the absence of a type. It is used to express that something (usually a function) has no value.

`Void` is a special case in the type system because it is not actually a type. It is used to express the absence of a type, which applies mostly to function arguments and return types. We have already "seen" Void in the initial "Hello World" example:

please review, doubled content

```
1 class Main {
2   public function Main():Void {
3     Debug.fTrace("Hello World");
4   }
5 }
```

The function type will be explored in detail in the section ?? (??) but a quick preview helps here: The type of the function `main` in the example above is `Void->Void`, which reads as "it has no arguments and returns nothing". Haxe does not allow fields and variables of type `Void` and will complain if an attempt at declaring such is made:

review please, sounds weird

```
1 // Arguments and variables of type Void are not allowed
2 var x:Void;
```

2.2 Nullability

Definition: nullable
A type in Haxe is considered nullable if `null` is a valid value for it.

It is common for programming languages to have a single, clean definition for nullability. However, Haxe has to find a compromise in this regard due to the nature of Haxe's target languages: While some of them allow and; in fact, default to `null` for anything, others do not even allow `null` for certain types. This necessitates the distinction of two types of target languages:

Definition: Static target
Static targets employ their own type system where `null` is not a valid value for basic types. This is true for the Flash, C++, Java and C# targets.

Definition: Dynamic target
Dynamic targets are more lenient with their types and allow `null` values for basic types. This applies to the JavaScript, PHP, Neko and Flash 6-8 targets.

for starters...please review

There is nothing to worry about when working with `null` on dynamic targets; however, static ones may require some thought. For starters, basic types are initialized to their default values.

Definition: Default values
Basic types have the following default values on static targets:

`Int`: `0`

`Float`: `NaN` on Flash, `0.0` on other static targets

`Bool`: `false`

As a consequence, the Compiler does not allow the assignment of `null` to a basic type on static targets. In order to achieve this, the basic type has to be wrapped as `Null<T>`: