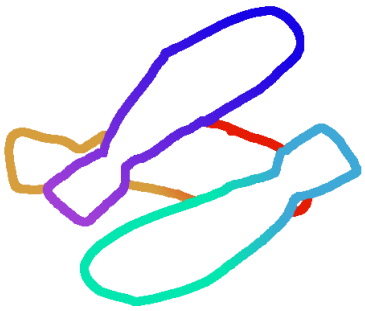

Carp Reference Documentation

Made by Melodi

Version 1.0.0

Learning Carp

This is a reference manual and instruction guide filled with samples and explanations learning Carp.

Getting started

Installation

1. Ensure dotnet is installed
2. Run the following to install it from NuGet

```
$ dotnet tool install --global carp
```

3. Check carp is installed

```
$ carp --version
```

4. It is recommended to use Visual Studio Code with the Carp Language extension to aide with development

Hello world

Create a new file, give it the .carp extension, and write the following code:

```
import std.io

IO.println('Hello, world!')
```

Now run the file with the Carp interpreter:

```
$ carp hello.carp
```

You should see the output `Hello, world!` printed to the console.
Alternatively you can go file-less and use the REPL:

```
$ carp
 : import std.io
 :
 : IO.println('Hello, world!')
```

Basics

Types

Primitive types

```
# Strings are only denoted with single quotes
str my_string = 'Hello, world!'

# Booleans are either true or false
bool my_bool = true

# Integers are whole numbers, decimals, and negative numbers
int my_int = 10
int my_decimal_int = -10.5

# Obj is the base type for all objects
obj my_obj = 5

# Null is the absence of a value
obj my_null = null

# Objects are automatically assigned null if not given a value
obj my_auto_null
# value is null

# If the object is a struct though, they will be auto-instatiated
int my_auto_int
# value is 0

# Let implicitly determines the type of the variable based on the value
let my_auto_typed_int = 5
```

Compound types

```
# Collections are a list of objects, they are denoted with *
int* my_list = [1, 2, 3, 4, 5]

# Maps are a collection of key-value pairs
str:int my_map = [
    'key1': 1,
    'key2': 2,
    'key3': 3
]

# Ranges are a collection between two values
range<int> my_range = 1..10
```

Type behaviours

Object `obj`

This is the base object of everything.

String `str`

Strings are immutable, meaning they cannot be changed after they are created. They are represented by surrounding the text value with single quotes `'hello world'`. Unlike other languages, double quotes are **not** permitted. They can use backslashes to escape a quote character or another backslash.

Casting:

- From any type to `str` will perform override `.string` on it
- `int` Converts the string into an integer
- `chr` Converts the string into a character, using the first character only
- `bool` Converts the string into a boolean, comparing it to 'true' and 'false'

Properties:

- `.length` `int` Calculates the length of the string
- `.lower` `str` Converts the string to lowercase
- `.upper` `str` Converts the string to uppercase
- `.clean` `str` Removes leading and trailing whitespace

- `.split(str delimiter) str*` Splits the string into an array of strings based on a delimiter
- `.replace(str source, str replacement) str` Replaces all occurrences of a substring with another substring
- `.contains(str search) bool` Checks if the string contains a substring
- `.startswith(str prefix) bool` Checks if the string starts with a substring
- `.endswith(str suffix) bool` Checks if the string ends with a substring

Overrides:

- `iterate` String overrides the iterate method to allow enumeration of each character in the string
- `add` String overrides the add method to concatenate two strings
- `multiply` String overrides the multiply method to repeat the string a specified number of times
- `divide` String overrides the divide method to split the string into a collection of strings based on a delimiter, which can be a `str` or a `chr`
- `index` String overrides the index method to get the character at a specified index, if a range is passed, it will return a substring based on that range

Character `chr`

Characters are represented by using a backtick and then a single character `` \a` ``.

Casting:

- `int` Converts the character into an integer

Properties:

- `.lower chr` Converts the character to lowercase
- `.upper chr` Converts the character to uppercase
- `.is_upper bool` Checks if the character is uppercase
- `.is_lower bool` Checks if the character is lowercase
- `.is_digit bool` Checks if the character is a digit
- `.is_alpha bool` Checks if the character is a letter
- `.is_symbol bool` Checks if the character is a symbol
- `.is_whitespace bool` Checks if the character is whitespace

- `.is_upper` `bool` Checks if the character is uppercase
- `.is_lower` `bool` Checks if the character is lowercase

Integer `int`

Integers are whole numbers, decimals, and negative numbers. They are represented by a number, and can be negative or decimal. They can be used in mathematical operations, and are represented by `5`, `-5`, `5.5`. They are a `struct` type which means they default to 0 instead of null.

Casting:

- `chr` Converts the integer into a character

Overrides:

- `add` Integer overrides the add method to add two integers
- `subtract` Integer overrides the subtract method to subtract two integers
- `multiply` Integer overrides the multiply method to multiply two integers
- `divide` Integer overrides the divide method to divide two integers
- `modulo` Integer overrides the modulo method to get the remainder of two integers
- `pow` Integer overrides the power method to raise the integer to the power of another integer
- `greater` Integer overrides the greater method to check if the integer is greater than another integer
- `less` Integer overrides the less method to check if the integer is less than another integer
- `negate` Integer overrides the negate method to get the inverse of the integer
- `step` Integer overrides the step method to get the current integer plus 1
- `iterate` Iterator from 0 to the integer

Boolean `bool`

Booleans are either true or false. They are represented by `true` or `false`. They can be used in logical operations, and are a `struct` type which means they default to `false` instead of null.

Overrides:

- `not` Boolean overrides the not method to get the inverse of the boolean

Function `func`

Functions are a type of object that can be called. They are represented by `func<treturn>` where the generic type is the return type of the function. They can be called by using the `()` operator.

Overrides:

- `call` Function overrides the call method to execute the body of the function

Collection `obj*`

Collections are a list of objects, they are represented by `[obj0, obj1, obj2]`. They can be accessed by index, and can be iterated over. When a collection is not supplied with a type, it will default to the highest common type, if one exists, else `obj*`

Casting:

- `tvalue*` Converts the collection into a new collection, if it the item's type extends the sub_type

Properties:

- `.length` `int` Calculates the length of the collection
- `.first` `tvalue` Gets the first item in the collection
- `.last` `tvalue` Gets the last item in the collection
- `.append(tvalue item)` `void` Appends an item to the end of the collection
- `.remove(tvalue item)` `void` Removes an item from the collection
- `.removeat(int index)` `void` Removes an item from the collection at a specified index
- `.insert(int index, tvalue item)` `void` Inserts an item into the collection at a specified index
- `.contains(tvalue item)` `bool` Checks if the collection contains an item
- `.within(int index)` `bool` Checks if the index is within the bounds of the collection
- `.clear()` `void` Clears the collection

Overrides:

- `add` Collection overrides the add method allow merging two collections of the same type
- `iterate` Collection overrides the iterate method to allow enumeration of each item in the

collection

- `index` Collection overrides the index method to get the item at a specified index, if a range is passed, it will return a sub-collection based on that range

Map `obj:obj`

Maps are a collection of key-value pairs, they are represented by `['key0': value0, 'key1': value1]`. They can be accessed by key, and can be iterated over. It uses the same high common type rule as collections, if no type is supplied.

Properties:

- `.length` `int` Calculates the length of the map
- `.keys` `tkey*` Gets all the keys in the map
- `.values` `tvalue*` Gets all the values in the map
- `.remove(tkey key)` `void` Removes an item from the map
- `.contains(tkey key)` `bool` Checks if the map contains a key
- `.clear()` `void` Clears the map

Range `range`

Ranges are represented as `start..end`, they must be the same type and the object must override `Less`, `Subtract` and `Step`. As a generic type, their type is referenced as `range<tvalue>`

Casting:

- `tvalue*` Converts range into a collection of type

Properties:

- `.length` `int` Calculates length of range based on `end` - `start`
- `.start` `tvalue` Start of range
- `.end` `tvalue` End of range

Overrides:

- `iterate` Range overrides the iterate method to allow enumeration between the start and the end points

Type `type`

Types are used to store the type of an object, they are represented by `t(obj)`. They can be used to check the type of an object, and can be used in casting.

Casting:

- From any type to `type` will return the type of the object

Properties:

- `.name` `str` Gets the name of the type
- `.base` `type` Gets the base type of the type
- `.is_struct` `bool` Checks if the type is a struct

Comments

Comments are used to annotate code, they are ignored by the interpreter.

```
# This is a comment

#(
   This is a multi-line comment
)
```

Make sure a space is present after the `#` symbol, otherwise the preprocessor will get confused with a directive.

Blocks

Blocks are used to group statements together, there are two types of blocks in Carp: the shorthand block and the full block.

```
# Shorthand block
int add(int a, int b) → a + b

# Full block
int add(int a, int b) {
    return a + b
}
```

Casting and handling types

Casting is used to convert one type to another, and type checking is used to determine the type of an object.

```
# Object type coercion, this is implicit type conversion
int my_num = '5'
# value is 5

# Explicit type conversion
int my_num = '5' ~ int
# value is 5

# Type checking
bool is_int = my_num ~~ int
# value is true

# Type storage
type my_type = t('hi')
# value is str
```

Operators

Operators are used to perform operations on values, such as addition, subtraction, and comparison.

```
# Arithmetic operators
int add = 5 + 5
int subtract = 5 - 5
int multiply = 5 * 5
int divide = 5 / 5
int modulo = 5 % 5
int power = 5 ^ 5

# All of these operators can be combined with the assignment operator to
modify the value of a variable
int num = 5
num += 5

# Comparison operators
bool equal = 5 == 5

bool not_equal = 5 ≠ 5
```

```
# or alternatively
not_equal = 5 <> 5

bool greater_than = 5 > 5
bool less_than = 5 < 5
bool greater_than_or_equal = 5 ≥ 5
bool less_than_or_equal = 5 ≤ 5

# Logical operators
bool and = true & false
bool or = true | false

bool not = !true
bool inverse = -num
```

Control flow

Control flow is used to determine the flow of the program, such as loops and conditionals.

```
# If statements
if 5 > 4 → IO.println('5 is greater than 4')

# If-else statements
if 5 > 4 → IO.println('5 is greater than 4')
else → IO.println('5 is not greater than 4')

# While loops
int i = 0
while i < 5 {
    IO.println(i)
    i += 1
}

# For loops
for i : 0..5 {
    IO.println(i)
}

# The 0..5 can actually be simplified to ..5
# But since for requires a collection, it will even take just 5
```

```
# For loops without storing the index
for 0..5 -> IO.println('hi')
```

Functions and methods

Functions are used to encapsulate code into reusable blocks, they can take arguments and return values.

```
# The return type is specified first, or if none, void
int add(int a, int b) -> a + b
void print(str message) -> IO.println(message)

# The arguments and their types are specified in the parentheses
# The block is specified directly after the arguments
```

Classes and structs

Classes are used to define objects with properties and methods, they can be instantiated and used in the program.

```
class MyClass {
    # Properties are defined the same way as variables
    int my_prop

    # Methods are defined in the same way as functions
    void print_prop() -> IO.println(this.my_prop)

    # Constructors are used to initialize the object
    void init(int prop) -> this.my_prop = prop
}

# Instantiating a class
MyClass my_obj = MyClass.new(5)

my_obj.print_prop()
```

Structs

Structs are similar to classes, but they cannot be null, meaning they must have an empty constructor for implicit instantiation

```
struct Vec2 {
    int x
    int y

    void init() {
        this.x = 0
        this.y = 0
    }

    void init(int x, int y) {
        this.x = x
        this.y = y
    }

    str string() -> 'Vec2(' + this.x + ', ' + this.y + ')'
}
```

Shortcuts

Winding and filtering allow applying an operation to a sequence of values. As of the current version, filtering is not supported, but uses the `;;` operator. Winding can be done using the `::` operator.

```
int nums = [1, 2, 3, 4, 5]
# this can actually be done using int* nums = 1..6
# or 1..6 ~ int* if type coercion is disabled

int new_nums = nums :: * 10
# value will be [10, 20, 30, 40, 50]
```

The winding operator `::` converts a collection into a "winded" object where all operations applied to it, will be applied to all items. This includes operators and properties, note that only one operation can be applied per wind.

```
str* names = ['John', 'Jane', 'Francis', 'Joe']

str* lower_names = names :: .lower
# value is ['john', 'jane', 'francis', 'joe']
```

As of the latest version, `bool` operators cannot be used with winded expressions and produce a collection, instead they will produce a single value, based on whether they all matched `true`.

The standard library and imports

Imports are used to include external code into the program, such as the standard library or other scripts.
They are controlled by the active package resolver. The `std` prefix is available to access the standard library. The `git` prefix is available to access the git package resolver. Not specifying one of these will search the current active directory or the current package's source.

```
# Standard library import
import std.io

# Github import
import git.username.repo

# Local import
import utils
```

IO

The IO package is responsible for input and output operations in Carp. It contains functions for controlling the connected console.

▶ Reference

Math

Math is for extended mathematical operations, not included in the base language's operators. It also contains random number generation functions.

▶ Reference

FS

The FS package is for file system operations, such as reading and writing files.

▶ Reference

Parse

Parse is a universal package for parsing JSON, XML, HTML and for Regex operations.

▶ Reference

Net

Net is for network operations, such as HTTP requests.

▶ Reference

Resource

Resource is for loading and managing external resources (the ones stored in /resources), such as text files.

▶ Reference

System

System is for system operations, such as executing shell commands.

▶ Reference

Projects

Projects are designed for larger codebases, they may contain multiple files, and have resources stored within them. They can later be packaged into `.caaarp` files for distribution and execution.

Creating a project

To create a new project, run the following command:

```
$ carp new -n my_project
```

Or you can convert an existing script into a project using:

```
$ carp new -s my_script.carp
```

Optionally the name of the project can be specified with the `-n` flag.

Running a project

To run a project, navigate to the project directory and run:

```
$ carp .
```

This will build and run the project. To just build it, use:

```
$ carp build .
```

These will generate a `.caaarp` file in the /export directory, named the projects name suffixed with the current version in the `.carpproj` file.

Project structure

A project will have all its scripts in the initial directory, and resources in the /resources directory. The /export directory will contain the generated `.caaarp` files.

Project configuration

The project configuration is stored in a `.carpproj` file in the project directory. It contains the project name, version, and other metadata.

Advanced

Interpreter flags

```
  -i, --interactive        Start the Carp REPL after executing the script.

  -c, --line               Execute the Carp code provided in the command line.

  -v, --verbose            (Default: true) Print the result of the script
execution to the console.

  -d, --debug              Start the Carp debugger.

  -f, --force-throw        Force the internal errors to trigger the native
stacktrace.

  --strict-warnings        Treat warnings as errors

  --default-non-structs    (Default: true) Allow non-struct objects to be
auto-initialized
```

```
  --implicit-casts          (Default: true) Enable implicit casting/type
coercion

  --help                    Display this help screen.

  --version                 Display version information.

  Path (pos. 0)             The path to the script, project or package to run.
```

The preprocessor

The preprocessor is responsible for formatting the input code for the lexer and parser. It adds support for directives, strips comments, re-writes imports and can be expanded in the future to do even more.

Directives

Include

The include directive, directly includes the specified file path in the current file. Note that this is not package-safe and will only use the current directory as a source, which makes it helpful for the REPL. It will also be preprocessed before being injected.

```
#include header.carp
```

Define

Define allows you to create macros, which are similar to methods except they directly replace tokens and their arguments with the body. Brackets can be used to allow it to span multiple lines

```
#define add(a, b) = (
    a + b
)

int sum = add(1, 2)
# This will be replaced with 1 + 2
```

It can also be used for things such as constants, as it does not need to take parameters

```
#define my_const = 25.5

my_const
# Replaced with 25.5
```

Style guide

Block bodies

The shorthand block body should be used whenever possible (whenever the body is a single expression).

```
# Single-line block body
int add(int a, int b) → a + b

# Multi-line block body
int slowmultiply(int a, int b) {
    int result = 0
    for b → result = add(result, a)
    return result
}
```

Naming conventions

Variables and properties are named in snake_case. Classes are named in upper CamelCase. Methods and functions are lower mergecase.

For example:

```
int my_var = 10

void print(str message) → IO.println(message)

class MyObj {
    void dosomething() → print('Hello, world!')
}
```