Blockly C++ Documentation

Chris Haidas (SSA)

Jonathan White

Christopher Lindberg

[**Preface**](#_o5olwg217kx) **3**

[**File Structure**](#_khdksxoy4pjf) **4**

[**Internal Logic**](#_kpsbcthpu3rd) **5**

[**C\_Scope**](#_mltodjuaip52) **6**

[Variables](#_oqvksaqrdqu5) 6

[Functions](#_fd9uzxylgq9t)8

[**C\_Logic**](#_4kjidw2m40ey) **14**

[Functions](#_shsysf2hkd10) 15

[**Coding Style**](#_s3wks7amlpz2) **16**

[Mutator Functions](#_5mta5gm2q9z) 17

[Non-Mutator Functions](#_ozy012d6ed46)19

[**New Code**](#_78bm7zq0ftxg) **22**

[Variables](#_isa57a12ynk1) 22

[Functions](#_2cok2ns3dnkq) 22

[**New Workspace Features**](#_7sfxatnoz9kr) **24**

[Functions](#_z20sttwe0f2d) 24

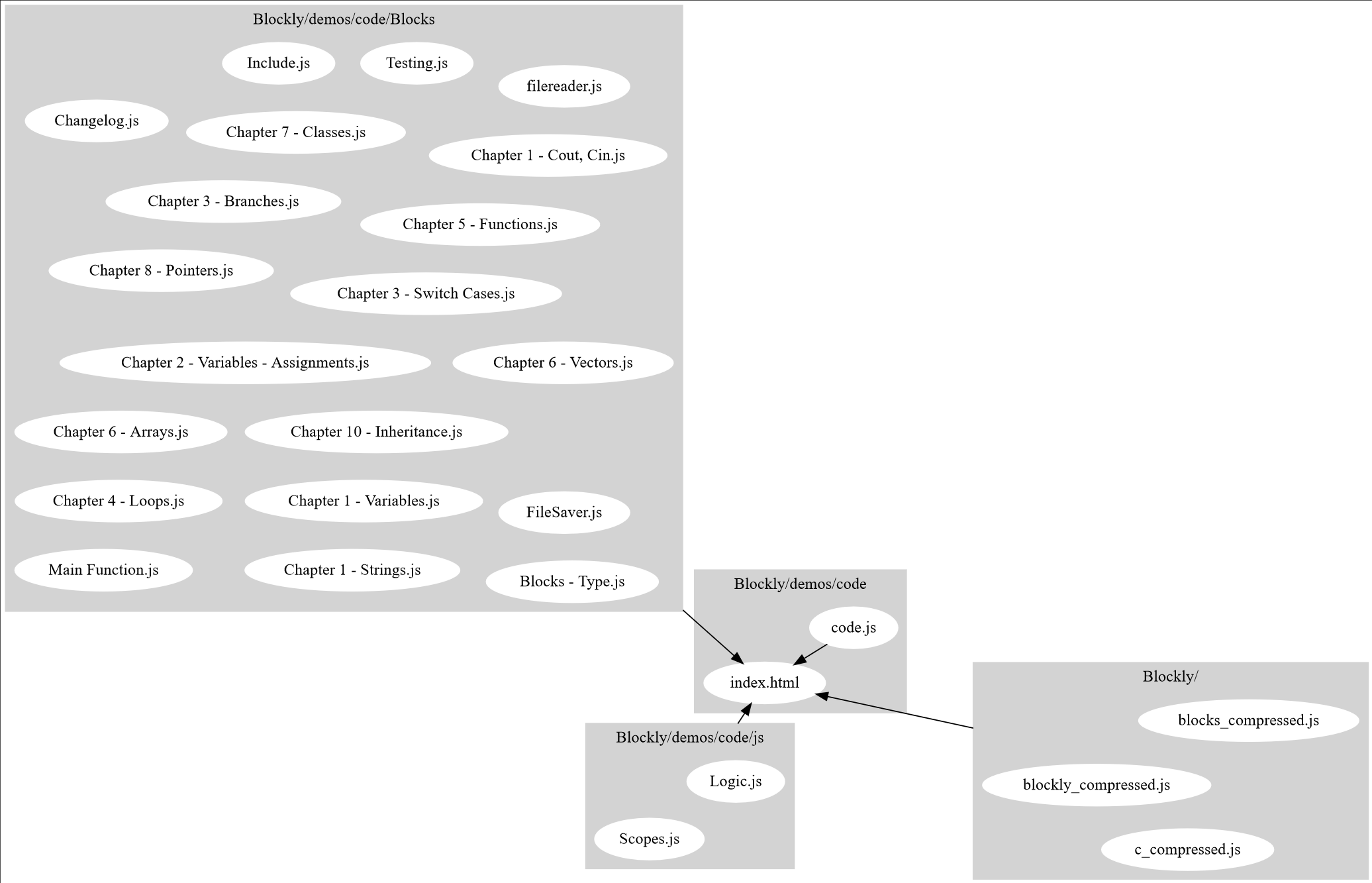
# Preface

Blockly C++ is a software developed by Chris Haidas, Jonathan White, and Christopher Lindberg that allows users to interact with “blocks” to generate code and to check types, logics and scopes. It is a software that is continuously being developed. Thus, both this documentation and the software are works in progress and will be updated at a later time.

The Blockly C++ uses the Blockly library; an API originally developed by Google software developer Neil Fraser and the Blockly team[[1]](#footnote-0). The Blockly Library is expansive and is freely available online[[2]](#footnote-1). Thus, this documentation does not serve to document the aforementioned library. Rather, this serves as a guide for the coding style, new code, and internal logic that exists in Blockly C++;

# File Structure

Like Blockly proper, Blockly C++ centralizes all code into an html file. Currently, there exists only one html workspace, so to speak, that is being worked on.



The map of files, as generated by graphviz

# Internal Logic

As a derivative, blocks in Blockly C++ are organized using N-ary Trees, that is, a data structure in which there may exist any number of child nodes derived from a given node. Therefore, a majority of the supplemental code exists to aid navigation through the Tree in order to aggregate and organize the data we need at any given time.

Each block is a “node” in the tree. Like nodes, each block in the workspace can have information that we need in order to perform a certain thing. A block that lets us declare a function has that unique property. As we search through the tree this property needs to be checked, if this property is the one we require (that is, the ability to create functions) then we grab the data from this block (as we would with a node). In Blockly C++, this is currently the primary method used for Scope checking. Thus, the object used to perform this is called *C\_Scope.*

# C\_Scope

## Variables

log:

Type - array<string>

Description - An array to log the IDs of blocks that have already been counted.

blockName:

Type - array<string>

Description - Names of the blocks that have been counted.

varName:

Type - array<string>

Description - Names of variables if the block is a creator block, such as declaring a variable or function.

objVar:

Type - array<string>

Description - List of variable declarations within this scope.

objFunc:

Type - array<string>

Description - List of function declarations within this scope.

dataStr:

Type - array<string>

Description - List of function declarations within this scope.

errorDef:

Type - string

Description - The variable or function declaration that is part of an error, e.g. a redeclaration.

## Functions

getLog():

Description - Getter for log.

Parameters:

Return - Returns variable log.

Return Type - null/array<string>

getBlockName():

Description - Getter for blockName.

Parameters:

Return - Returns variable blockName.

Return Type - null/array<string>

getVarName():

Description - Getter for varName.

Parameters:

Return - Returns variable blockName.

Return Type - null/array<string>

getObjVar():

Description - Getter for objVar.

Parameters:

Return - Returns variable objVar.

Return Type - null/array<string>

getObjFunc():

Description - Getter for objFunc.

Parameters:

Return - Returns variable objFunc.

Return Type - null/array<string>

getDataStr():

Description - Getter for objFunc.

Parameters:

Return - Returns variable objFunc.

Return Type - null/array<string>

getError():

Description - Getter for error.

Parameters:

Return - Returns variable error.

Return Type - null/string

create\_node\_log(node):

Description - Function to push the information we need from any block during the traversal. Calls validate\_node\_log to check if this node has already been checked. If it has, it pushes the information we require.

Parameters:

* Node - The block in the workspace that we’re currently looking at. (Blockly.Block)

Return -

Return Type -

validate\_node\_log(node):

Description - Checks the list of IDs to see if it exists. If it does, then it is checking a node that has already been checked and returns true.

Parameters:

* Node - The block in the workspace that we’re currently looking at. (Blockly.Block)

Return - If array log already has the id of the block we’re checking.

Return Type - boolean

recursion\_log(node, clear):

Description - The central function that traverses through the tree of blocks using recursion. This is likely the first function used in C\_Scope.

Parameters:

* Node - The block in the workspace that we’re currently looking at. (Blockly.Block)
* Clear - If the data arrays should be cleared. (boolean)

Return -

Return Type -

obj\_type\_log(node):

Description - Checks if the block we’re looking at is defining the data we want, e.g. a function declaration or a variable declaration. If it is, it pushes the data into their respective arrays. Used to count the variable/function declarations in the scope.

Parameters:

* Node - The block in the workspace that we’re currently looking at. (Blockly.Block)

Return -

Return Type -

data\_str\_log(node):

Description - Function to get the data structure type, e.g. if this block declares a function/variable etc.

Parameters:

* Node - The block in the workspace that we’re currently looking at. (Blockly.Block)

Return -

Return Type -

check\_unique\_array():

Description - Checks to see if there are repeating declarations in a given scope.

Parameters:

Return - The first element is whether there exists an error. If [0] is true, [1] returns the first location of the declared variable/function, etc. [2] returns the second location in this tree scope. If [0] is false, [1] and [2] both return -1.

Return Type - array<Boolean, Number>

# C\_Logic

Within the logic of every declared block, there are certain functions that either occur globally (such as using\_namespace\_std) or simply require the same logic (such as helper functions). Thus, the goal of C\_Logic is currently twofold. One, to keep track of the global declarations, such as if a library has been included. Second, to allow an easy use of helper functions instead of rewriting the same code *ad nauseum.* At this time, much of the *C\_Logic* object has not been fully implemented with the exception of the function *is\_element\_unique()*.

## 

## Functions

is\_element\_unique(arr, uni):

Description - Checks if an array has an element that occurs more than once. If uni is undefined, the function checks the given array to see if there are any repeating elements. If uni is defined, then the function checks if uni occurs more than once within arr.

Parameters:

* Arr - The array that we’re checking. (array<string>)
* Uni - An element that we’re specifically looking for. (string)

Return - If the element occurs more than once within the given array.

Return Type - boolean

# Coding Style

*Coding Style* specifically refers to the typical formatting practices used while declaring and generating new blocks. This is not a guide on how to declare blocks, but rather how block information is partitioned instead of all block code being within the same function, even if the functions themselves are defined by the standard Blockly Library.[[3]](#footnote-2)

Init():

Description - The first part of a block that runs. This function is used for when a block first loads. Unless called manually, this function will only run once on any given block. This is where the core information on any given block will initialize.

Parameters:

Return -

Return Type -

## Mutator Functions

Secondly, there are the mutators. As mentioned before, mutators are defined within the Blockly Library, but this refers to *how* we use and partition the information within the mutators, rather than the mutators themselves.

mutationToDom():

Description - The first function of a mutator. This saves mutation data by accessing the blocks within the mutators.

Parameters:

Return - container - The attributes we define.

Return Type - DOM

domToMutation(xmlElement):

Description - The second part of a mutator. Used to gather information (such as if the mutator block makes the main block constant) into the current block.

Parameters:

* xmlElement- The xml of the from the general DOM. (DOM)

Return -

Return Type -

decompose(containerBlock):

Description - The third part of a mutator. Called when the mutator box opens, used to typically set information, and used to populate the mutator box with blocks that can be used.

Parameters:

* containerBlock- The workspace of the mutator box. (Blockly.Block)

Return - containerBlock

Return Type - Blockly.Block

compose(containerBlock):

Description - The fourth part of a mutator. Called when the mutator box closes, used to modify the main block.

Parameters:

* containerBlock- The workspace of the mutator box. (Blockly.Block)

Return - containerBlock

Return Type - Blockly.Block

saveConnection(containerBlock):

Description - The fifth but optional part of a mutator. Called when the main block changes. Used to ensure the correct inputs maintain the correct connections with any block connecting to this current one.

Parameters:

* containerBlock- The workspace of the mutator box.

Return -

Return Type -

## Non-Mutator Functions

The following are functions that are not used for mutators.

onchange():

Description - A function that is called automatically by the Blockly Library, such as when a user opens the toolbox, moves the block, etc. onchange() is typically used to update characteristics on the main block, and to call other non-mutator functions.

Parameters:

Return -

Return Type -

allocateVariables():

Description - A function typically used to aggregate variable information for the current block. Inheritance mainly utilizes this function to allocate variables declared with a class/struct, then push it into the current block for scope checking.

Parameters:

Return -

Return Type -

allocateWarnings():

Description - A function used to deal with block warnings. Variable TT is declared as a string. Then, the function checks for any possible warnings/errors/block errors that may occur, concatenates it into variable TT. TT is used because multiple setWarningText(TT) functions cannot be used. Thus, these type and logic checks concatenate errors into TT, then TT is used as a formatted string for display. If TT has a length of more than 0, the function sets the warning text to TT. Else, it sets the warning text to null.

Parameters:

Return -

Return Type -

allocate…():

Description - While not a function *per se,* allocate…() is a style often used to help partition the code. “...” refers to any objective of the code, such as allocateWarnings(), or allocateVariables().

Parameters:

Return -

Return Type -

# New Code

*New code* refers to any additions to the Blockly Library itself. Due to the Library being so massive, very few additions were made.

## Variables

dataStr:

Type - object of boolean variables

Description - Boolean checks to see if this block is a variable, array, vector function, struct or class.

## Functions

setFalse():

Description - Used to set all variables within the dataStr object to false.

Parameters:

Return -

Return Type -

setDataStr(obj, bool):

Description - Used to set all variables within the dataStr object to false.

Parameters:

* Obj - Used to set the type of data structure for a particular block. (string)
* Bool - Used to call Blockly.Block.setFalse() in order to set/reset the type of block. (boolean)

Return -

Return Type -

getDataStr():

Description - Returns the type of data structure type the block is.

Parameters:

Return -

Return Type - String

# New Workspace Features

## Functions

The following code is for the Blockly workspace that the user may interact with, including the ability to download code, save, and load.

downloadCode():

Description - Downloads the C++ generated code into a .cpp file. This file is named main.cpp.

Parameters:

Return -

Return Type -

downloadXML():

Description - The saving feature; this function downloads the block XML into a txt file. The name is “blockly save “, then the current hour, minute and second (to disambiguate saves and to prevent save overwriting).

Parameters:

Return -

Return Type -

readFile(input):

Description - The loading feature; this function uploads a blockly save into the current workspace. It wipes out all current blocks on the workspace and reloads inputs the blocks as they were during saving.

Parameters:

* Input - The file to read from. (.txt file)

Return -

Return Type -

downloadXML():

Description - The saving feature; this function downloads the block XML into a txt file. The name is “blockly save “, then the current hour, minute and second (to disambiguate saves and to prevent save overwriting).

Parameters:

Return -

Return Type -

1. N. Fraser, "Ten things we've learned from Blockly," 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond), Atlanta, GA, 2015, pp. 49-50. [↑](#footnote-ref-0)
2. Introduction to Blockly | Google Developers. (n.d.). Retrieved from https://developers.google.com/blockly/guides/overview [↑](#footnote-ref-1)
3. For an in depth look at the mutators and their general purpose, see “Mutators | Blockly | Google Developers.” *Google*, Google, developers.google.com/blockly/guides/create-custom-blocks/web/mutators [↑](#footnote-ref-2)