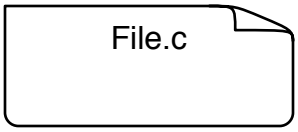
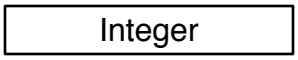
Code Storage



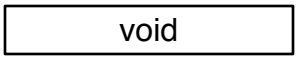
A **file** stores the source code for all classes and methods. It appears as a rectangle with a corner turned, resembling file icons. Its diagram will display all classes and functions declared, but will omit implementations. If this leaves the file empty, an explanation in italics and parenthesis will be given.

Data Types

C values describe any non-function data type that can be purely expressed in C. They appear as plain text inside a sharp-edged shadowless rectangle.



A **primitive** is a scalar or float, such as bool, char, int, float, and double.

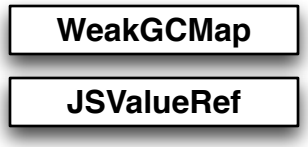


A **pointer** stores a memory address to one or more other values. One common type is the **void * pointer**, which can point to non-C objects.

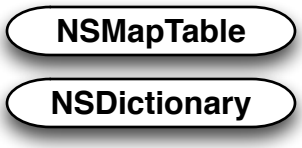
A **compound value** is a collection of other values grouped as a single value. If the other share memory, it is a **union**. Otherwise, it is a **C structure**, abbreviated as a C struct.

A **C function** describes a function that follow C or C++ naming. They appear as plain text inside a shadowless pair of brackets.

Class Objects

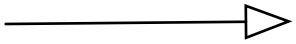


A **C++ object** is an instantiation of a class written in C++. It appears as bold text inside a shadowed sharp-edged rectangle. The text will be the name of the class with a namespace, the class without a namespace, a pointer datatype, or a common abbreviation.



An **Objective-C object** is an instantiation of a class written in C++. It appears as bold text inside a shadowed rounded rectangle. The text will be the name of the class, a Core Foundation datatype equivalent, or a common abbreviation.

Ownership & Relations

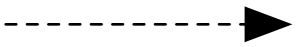


An **opaque relationship** is indicated by a solid line with an open (white) arrow pointing to an **opaquely held** object or value. It is the storage of an address or value with no memory management performed whatsoever.

- **Does not** ensure object will remain
- **Does not** clean up when object is removed
- **Is not** safe to look at object beyond comparing addresses
- **Will not** cause retain cycles in non-GC languages

A **strong relationship** is indicated by a solid line with a closed (black) arrow pointing to a **strongly held** object. It follows memory retention rules of the pointed object to ensure it is not collected or deallocated.

- **Does** ensure object will remain
- **Does** clean up when object is removed
- **Is** safe to look at object beyond comparing addresses
- **May** cause retain cycles in non-GC languages



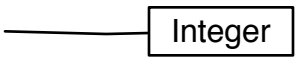
A **weak relationship** is indicated by a dashed line with a closed (black) arrow pointing to a **weakly held** object. It follows weak reference rules of the pointed object and is broken when it is collected or deallocated.

- **Does not** ensure object will remain
- **Does** clean up when object is removed
- **Is** safe to look at object beyond comparing addresses
- **Will not** cause retain cycles in non-GC languages

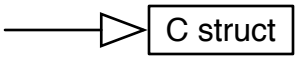
A **value relationship** is indicated by a solid line with no arrow connecting a value. It is for when, instead of storing a pointer (direct or indirect) to an object or value, the value itself is stored in the instance variable. There is no memory management, as this is to store primitive values.

- **Does** ensure value will remain
- **Does** clean up when object is removed
- **Is** safe to look at object beyond comparing addresses
- **Will not** cause retain cycles in non-GC languages

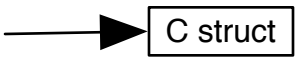
Relationship Examples



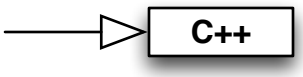
A **value relationship to an integer**, with the integer value into the pointer field. No memory management or derefrencing is done.



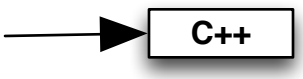
An **opaque relationship to a C datatype** does no memory management, and does not guarantee the object will hang around. Treat these with extreme caution, as usage of the held data may require a special situation.



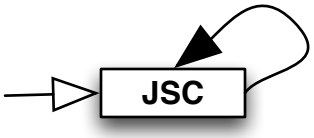
An **strong relationship to a C datatype** is done via the traditional C memory management of new and free.



An **opaque relationship to a C++ object** does no memory management, and does not guarantee the object will hang around. Treat the memory pointer as a value, not something to examine except in special conditions.

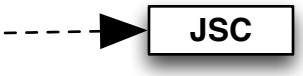


An **strong relationship to a Non-Javascript Core C++ object** is done via the traditional C++ memory management, of new and delete.

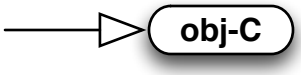


One such exception is when the **Javascript Core object is opaqely held and given the JSValueProtect call**, which will insure it will not get collected until a JSValueUnprotect is given. Here, the JSValueProtect is to simulate a strong relationship, and will be diagrammed as referring to itself.

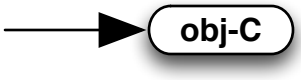
An **strong relationship to a Javascript Core object** is a signal that the object should not be collected until the relationship is broken or the holder is collected itself. The difference between it and a JSValueProtect is that the latter may lead to retain loops, leaking objects.



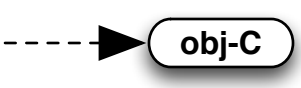
An **weak relationship to a Javascript Core object** will not stop the object from being collected. However, unlike an opaque relationship, the collection will trigger code that removes the relationship or other side effects.



An **opaque relationship to an Objective-C object** does no memory management, and does not guarantee the object will hang around. In classic Objective-C parlance, the pointer is assigned or __unsafe_unretained.



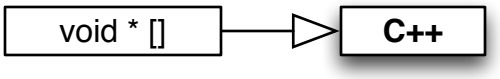
An **strong relationship to an Objective-C object** is done via a retain, via an explicit method call, via a synthesized property, or via ARC without a _weak keyword. The object will not be released until the relationship is broken, possibly by the referencer's own deallocation.



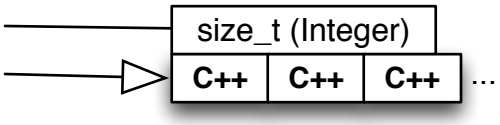
An **weak reference to an Objective C object** is done via ARC with a _weak keyword, or the underlying C functions. However, unlike an opaque reference, the deallocation will trigger code that removes the reference or performs other side effects.

Arrays

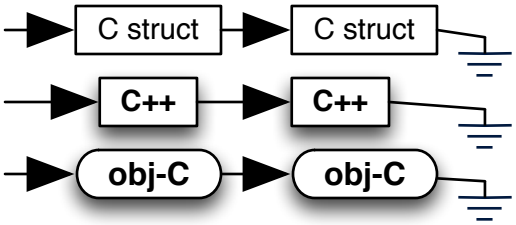
An **array**, for the purposes of these diagrams, refers to any data type (Or collection thereof) that is used to manage 0 or more of one datatype. Arrays may be able to specify a specific item by an index. This definition intentionally ignores requirements of implementation and is vague enough to allow for sets to be included in this category.



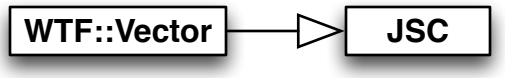
A **pointer array** is a classic C data type in which a contiguous collection of pointers are allocated and populated. It is meant as a short-term, low impact memory structure, pointing to C++ objects without any memory management.



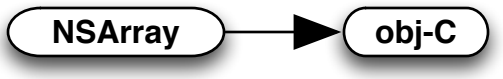
An **C array** is another short-term C array data type where a single pointer points to a contiguous collection of datatypes, with an integer specifying how many exist. Three such datatypes will be shown, followed by an ellipses (…) to indicate variable count.



A **linked list** is a design pattern that may be done by any datatype that has a pointer to another of its type, including C structures, C++ objects, or Objective C objects. The relationship is typically strong, and is represented as a chain of the datatype, with the end of the chain ending in a grounding symbol to indicate 'nil' or 'NULL'.



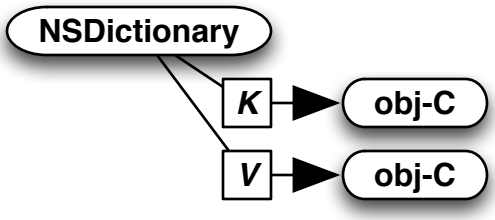
WTF::Vector<JSValueRef>, often shortened to **Vector**, is a C++ class that offers more flexibility than the pointer array, but still is used in places where the objects will not be garbage collected, and thus, no strong nor weak relationship is needed.



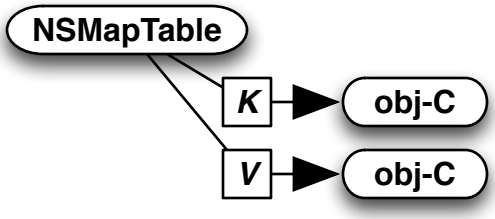
NSArray, CFArray, or **NSCFArray** are all the same class that hold strong relationships to Objective C objects as an array. While the CFArray variant can be customized to other relationship-datatype situations, it is rarely done in practice.

Maps & Dictionaries

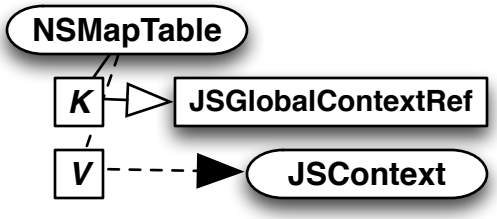
A **map**, for the purposes of these diagrams, refers to any data type that stores a 1:1 association of a value or object of one data type to a value or object of another data type. Given a key, there is a value associated. The keys must be unique, but the values are not so limited.



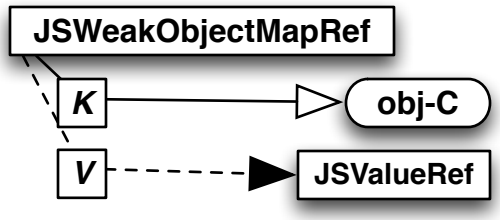
NSDictionary, CFDictionary, or **NSCFDictionary** are all the same Objective-C class containing key/value tuples in strong relationships to Objective C objects. The tuple is represented as a pair of relationships, one labeled K for key, the other V for value. While the CFDictionary variant can be customized to other relationship and datatype handling, it is rarely done in practice.



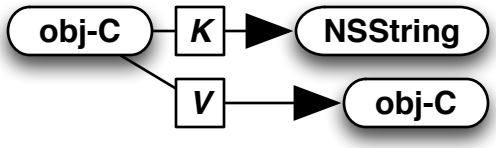
NSMapTable is an Objective-C class containing key/value tuples with customizable relationship and datatype handling. It allows for opaque relationships for all values, as well as strong and weak relationships with Objective C classes.



One common usage is a lookup table spanning between Javascript Core and an Objective C wrapper. As an example, this MapTable tracks JSGlobalContextRef to the JSContext that wrappers it. To do such, it has an opaque relationship to JSGlobalContextRef keys and weak relationship to JSContext. When the value is deallocated, the tuple is removed.



OpaqueJSWeakObjectMap and its pointer type, **JSWeakObjectMapRef** is a C++ class containing key/value tuples with an opaque void or object key and a weak JSValueRef value. It is defined further in its own page.



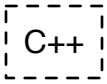
Due to Objective-C's **Key Value Coding**, all NSObjects can function as limited maps, but often has limitations on what keys are valid, and what manner of values can be stored.

Class Definitions

A **class definition** declares a class, its superclass, as well as aspects about the class. In Objective C, this class definition may be spread out across multiple files. It appears as a shadowed sharp-edged rectangle with various added shapes and text:

The **superclass** may optionally be displayed on top. It appears as plain text inside a shadowed roofed rectangle. Templates shall use empty brackets < >.

The **class type** indicates the language and origin of the class. It appears as plain text inside a shadowless shape with a dashed border in the upper right corner. For C++ classes, the shape is a sharp edged rectangle. For Objective-C classes, the shape is a rounded rectangle.



Generic C++ class

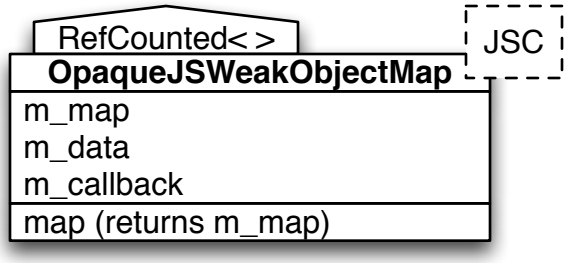
C++ class defined by the WTF library

C++ class defined by JavaScript Core

Objective-C class

Core Foundation or Foundation class

Objective-C class from the iOS SDK



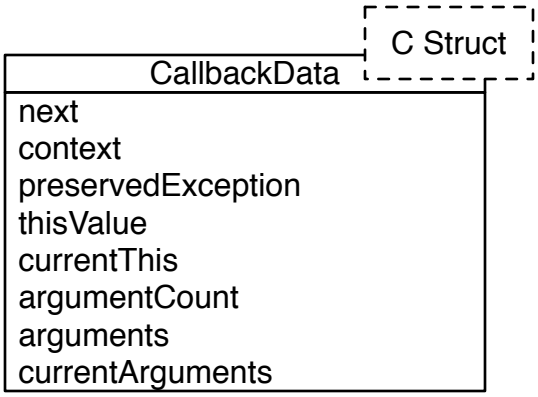
The **class name** appears in its own section, in bold text.

The **class attributes** are also known as member or instance variables and are in the next section. They may have relationships shown. Static attributes are not shown.

The **class operations** are also known as member functions or instance methods and are in the last section. Return values are mentioned in parenthesis after the name. Static functions or class methods are not shown.

C Structures

A **C structure**, mentioned before, is a compound datatype, that predates C++ and Objective-C classes and shares a common heritage. To that end, the metaphor and diagram is similar. Since this is a C datatype, it appears as a sharp-edged shadowless rectangle with plain text.



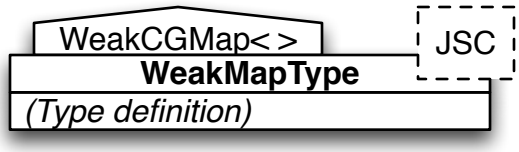
The **class type** uses a sharp edged, dashed-border rectangle. The text is "C Struct".

The **type name** appears in its own section, in plain text.

The **struct components**, similar to class attributes, are in the next section. They may have relationships shown.

C structs do not have any operations, member functions, or instance methods. They also have no superclass.

Type Definitions



A **type definition** is the creation of a new data type, either from another type, declaring a pointer type, or declaring an implementation of a template. It resembles the class definition.

The **class type** reflects the nature of the new data type, and follows the same rules as above.

The **previous class** reflects the datatype being redefined, and is placed where the superclass is.

The **class name** is the new datatype name, and is placed where the class name is.

Since no new attributes nor operations are given, those sections are replaced with the text "Type definition" in parenthesis and italics.