

---

# Objective-C 1 Runtime Reference

**(Not Recommended)**





Apple Inc.  
© 2002, 2009 Apple Inc.  
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.  
1 Infinite Loop  
Cupertino, CA 95014  
408-996-1010

Apple, the Apple logo, Cocoa, Mac, Mac OS, and Objective-C are trademarks of Apple Inc., registered in the United States and other countries.

Simultaneously published in the United States and Canada.

**Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

**IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY**

**DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.**

**THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.**

**Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.**

# Contents

## Objective-C 1 Runtime Reference (Not Recommended) 7

---

Overview	7
Functions by Task	7
Accessing Selectors	7
Sending Messages	8
Forwarding Messages	8
Adding Classes	9
Accessing Methods	9
Accessing Instance Variable Definitions	10
Accessing the Class Version	10
Posing As Another Class	11
Obtaining Class Definitions	11
Instantiating Classes	11
Accessing Instance Variables	12
Functions	12
class_addMethods	12
class_createInstance	12
class_createInstanceFromZone	13
class_getClassMethod	13
class_getInstanceMethod	13
class_getInstanceVariable	14
class_getVersion	14
class_nextMethodList	15
class_poseAs	15
class_removeMethods	16
class_setVersion	16
marg_free	17
marg_getRef	17
marg_getValue	17
marg_malloc	18
marg_setValue	18
method_getArgumentInfo	18
method_getNumberOfArguments	19
method_getSizeOfArguments	19
objc_addClass	19
objc_getClass	21
objc_getClassList	22
objc_getMetaClass	23
objc_lookUpClass	23
objc_msgSend	23
objc_msgSendSuper	24

- objc\_msgSendSuper\_stret 24
- objc\_msgSendv 25
- objc\_msgSendv\_fpret 25
- objc\_msgSendv\_stret 26
- objc\_msgSend\_fpret 26
- objc\_msgSend\_stret 27
- objc\_setClassHandler 27
- object\_getInstanceVariable 28
- object\_setInstanceVariable 28
- sel\_getName 28
- sel\_getUid 29
- sel\_isMapped 29
- sel\_registerName 29
- Callbacks 30
  - ClassHandlerCallback 30
- Data Types 30
  - Class-Definition Data Structures 30
  - Instance Data Types 36
  - Macros 37

---

**Document Revision History 39**

---

**Index 41**

---

# Listings

## Objective-C 1 Runtime Reference (Not Recommended) 7

---

- Listing 1      Obtaining class method definitions 15
- Listing 2      Creating an Objective-C class definition 20
- Listing 3      Using objc\_getClassList 22



# Objective-C 1 Runtime Reference (Not Recommended)

---

**Important:** The information in this document is obsolete and should not be used for new development.

## Overview

This document describes the Mac OS X Objective-C runtime library support functions and data structures for version 1 of the language. The functions are implemented in the shared library found at `/usr/lib/libobjc.A.dylib`. This shared library provides support for the dynamic properties of the Objective-C language, and as such is linked to by all Objective-C applications.

This legacy document is useful primarily for developing bridge layers between Objective-C and other languages, or for low-level debugging for versions of Mac OS X prior to 10.5. You typically do not need to use the Objective-C runtime library directly when programming in Objective-C.

For the current version of the runtime, see *Objective-C Runtime Reference* and *Objective-C Runtime Programming Guide*.

The Mac OS X implementation of the Objective-C runtime library is unique to the Mac OS X platform. For other platforms, the GNU Compiler Collection provides a different implementation with a similar API. This document covers only the Mac OS X implementation.

## Functions by Task

### Accessing Selectors

This section describes the Objective-C runtime functions that you can use to manipulate selectors. These functions are declared in `/usr/include/objc/objc.h`.

- [sel\\_getName](#) (page 28) returns the name of the method specified by the selector.
- [sel\\_isMapped](#) (page 29) indicates whether a selector is registered.
- [sel\\_registerName](#) (page 29) registers a method with the Objective-C runtime system, maps the method name to a selector, and returns the selector value.
- [sel\\_getUid](#) (page 29) registers a method name with the Objective-C runtime system.

[sel\\_getName](#) (page 28)

Returns the name of the method specified by the selector.

[sel\\_isMapped](#) (page 29)

Indicates whether a selector is registered.

[sel\\_registerName](#) (page 29)

Registers a method with the Objective-C runtime system, maps the method name to a selector, and returns the selector value.

[sel\\_getUid](#) (page 29)

Registers a method name with the Objective-C runtime system.

## Sending Messages

When it encounters a method invocation, the compiler might generate a call to any of several functions to perform the actual message dispatch, depending on the receiver, the return value, and the arguments. You can use these functions to dynamically invoke methods from your own plain C code, or to use argument forms not permitted by `NSObject`'s `perform...` methods. These functions are declared in `/usr/include/objc/objc-runtime.h`.

- [objc\\_msgSend](#) (page 23) sends a message with a simple return value to an instance of a class.
- [objc\\_msgSend\\_fpret](#) (page 26) sends a message with a floating point return value to an instance of a class (i386 only).
- [objc\\_msgSend\\_stret](#) (page 27) sends a message with a data-structure return value to an instance of a class.
- [objc\\_msgSendSuper](#) (page 24) sends a message with a simple return value to the superclass of an instance of a class.
- [objc\\_msgSendSuper\\_stret](#) (page 24) sends a message with a data-structure return value to the superclass of an instance of a class.

[objc\\_msgSend](#) (page 23)

Sends a message with a simple return value to an instance of a class.

[objc\\_msgSend\\_fpret](#) (page 26)

Sends a message with a floating-point return value to an instance of a class.

[objc\\_msgSend\\_stret](#) (page 27)

Sends a message with a data-structure return value to an instance of a class.

[objc\\_msgSendSuper](#) (page 24)

Sends a message with a simple return value to the superclass of an instance of a class.

[objc\\_msgSendSuper\\_stret](#) (page 24)

Sends a message with a data-structure return value to the superclass of an instance of a class.

## Forwarding Messages

This section describes the functions used by `NSObject` and `NSInvocation` to forward method invocations. The arguments to the method are given as a list of arguments, and as such the nature of the calling convention varies for each CPU architecture. To invoke a method normally or to call a superclass method implementation, use the functions described in “[Sending Messages](#)” (page 8). These functions are declared in `/usr/include/objc/objc-runtime.h` and `objc-class.h`.

- [objc\\_msgSendv\\_fpret](#) (page 25) sends a message with a simple return value.
- [objc\\_msgSendv\\_fpret](#) (page 25) sends a message with a floating-point return value (i386 only).
- [objc\\_msgSendv\\_stret](#) (page 26) sends a message with a data-structure return value.

- [marg\\_malloc](#) (page 18) allocates an argument list.
- [marg\\_free](#) (page 17) releases an argument list.
- [marg\\_getRef](#) (page 17) returns a pointer to an argument in an argument list.
- [marg\\_getValue](#) (page 17) returns the value of an argument in an argument list.
- [marg\\_setValue](#) (page 18) sets the value of an argument in an argument list.

[objc\\_msgSendv](#) (page 25)

Given an argument list, send a message with a simple return value.

[objc\\_msgSendv\\_fpret](#) (page 25)

Given an argument list, send a message with a floating point return value.

[objc\\_msgSendv\\_stret](#) (page 26)

Given an argument list, send a message with a data-structure return value.

[marg\\_malloc](#) (page 18)

Macro that allocates an argument list.

[marg\\_free](#) (page 17)

Macro that releases an argument list.

[marg\\_getRef](#) (page 17)

Macro that returns a pointer to an argument in an argument list.

[marg\\_getValue](#) (page 17)

Macro that returns the value of an argument in an argument list.

[marg\\_setValue](#) (page 18)

Macro that sets the value of an argument in an argument list.

## Adding Classes

The function [objc\\_addClass](#) (page 19) is used to register a class with the Objective-C runtime. This function is declared in `/usr/include/objc/objc-runtime.h`.

[objc\\_addClass](#) (page 19)

Registers a class definition with the Objective-C runtime.

## Accessing Methods

These functions allow you to access information about methods (they are declared in `/usr/include/objc/objc-class.h`):

- [class\\_getInstanceMethod](#) (page 13) returns a pointer to the data structure describing the specified instance method.
- [class\\_getClassMethod](#) (page 13) returns a pointer to the data structure describing the specified class method.
- [class\\_nextMethodList](#) (page 15) returns one of the method lists contained in the specified class definition.
- [class\\_addMethods](#) (page 12) adds a list of methods to a class definition.
- [class\\_removeMethods](#) (page 16) removes a previously added list of methods from a class definition.

- [method\\_getNumberOfArguments](#) (page 19) returns the number of arguments accepted by a method.
- [method\\_getSizeOfArguments](#) (page 19) returns the total size of the stack frame occupied by a method's arguments.
- [method\\_getArgumentInfo](#) (page 18) returns information about one of a method's arguments.

[class\\_getInstanceMethod](#) (page 13)

Returns a pointer to the data structure describing the specified instance method.

[class\\_getClassMethod](#) (page 13)

Returns a pointer to the data structure describing the specified class method.

[class\\_nextMethodList](#) (page 15)

Returns one of the method lists contained in the specified class definition.

[class\\_addMethods](#) (page 12)

Adds a list of methods to a class definition.

[class\\_removeMethods](#) (page 16)

Removes a previously added list of methods from a class definition.

[method\\_getNumberOfArguments](#) (page 19)

Returns the number of arguments accepted by a method.

[method\\_getSizeOfArguments](#) (page 19)

Returns the total size of the stack frame occupied by a method's arguments.

[method\\_getArgumentInfo](#) (page 18)

Returns information about one of a method's arguments.

## Accessing Instance Variable Definitions

The [class\\_getInstanceVariable](#) (page 14) method allows you to access information about instance variables. It's declared in `/usr/include/objc/objc-class.h`.

[class\\_getInstanceVariable](#) (page 14)

Obtains information about the instance variables defined for a particular class.

## Accessing the Class Version

These functions let you set and get the version of a class definition (they are declared in `/usr/include/objc/objc-class.h`):

- [class\\_setVersion](#) (page 16) sets the version number of a class definition.
- [class\\_getVersion](#) (page 14) returns the version number of a class definition.

[class\\_setVersion](#) (page 16)

Sets the version number of a class definition.

[class\\_getVersion](#) (page 14)

Returns the version number of a class definition.

## Posing As Another Class

The `class_poseAs` (page 15) function globally overrides a specified class definition with a different class definition. The method is declared in `/usr/include/objc/objc-class.h`.

`class_poseAs` (page 15)

Globally override a specified class definition with a different class definition.

## Obtaining Class Definitions

These functions give you access to class definitions (they are declared in `/usr/include/objc/objc-runtime.h`):

- `objc_getClassList` (page 22) obtains the list of registered class definitions.
- `objc_getClass` (page 21) returns a pointer to the class definition of the specified class.
- `objc_lookupClass` (page 23) returns a pointer to the class definition of the specified class.
- `objc_getMetaClass` (page 23) returns a pointer to the metaclass definition of the specified class.
- `objc_setClassHandler` (page 27) sets a custom error-handling callback function called from `getMetaClass` and `objc_getClass` when the desired class is not found.

`objc_getClassList` (page 22)

Obtains the list of registered class definitions.

`objc_getClass` (page 21)

Returns a pointer to the class definition of the specified class.

`objc_lookupClass` (page 23)

Returns a pointer to the class definition of the specified class.

`objc_getMetaClass` (page 23)

Returns a pointer to the metaclass definition of the specified class.

`objc_setClassHandler` (page 27)

Sets a custom error-handling callback function called from `objc_getClass` (page 21) and `objc_getMetaClass` (page 23) when the desired class is not found.

## Instantiating Classes

These functions let you create instances of classes (they are declared in `/usr/include/objc/objc-class.h`):

- `class_createInstance` (page 12) creates an instance of a class, allocating memory for the class in the default malloc memory zone.
- `class_createInstanceFromZone` (page 13) creates an instance of a class, allocating memory for the class in a specific malloc memory zone.

`class_createInstance` (page 12)

Creates an instance of a class, allocating memory for the class in the default malloc memory zone.

`class_createInstanceFromZone` (page 13)

Creates an instance of a class, allocating memory for the class in a specific malloc memory zone.

## Accessing Instance Variables

These functions give you access to the instance variables of an object (they are declared in `/usr/include/objc/objc-class.h`):

- [object\\_setInstanceVariable](#) (page 28) changes the value of an instance variable of a class instance.
- [object\\_getInstanceVariable](#) (page 28) obtains the value of an instance variable of a class instance.

[object\\_setInstanceVariable](#) (page 28)

Changes the value of an instance variable of a class instance.

[object\\_getInstanceVariable](#) (page 28)

Obtains the value of an instance variable of a class instance.

## Functions

### **class\_addMethods**

Adds a list of methods to a class definition.

```
void class_addMethods(Class aClass, struct objc_method_list* methodList)
```

#### Parameters

*aClass*

A pointer to an [objc\\_class](#) (page 31) data structure. To add instance methods, pass the class definition to which you wish to add the methods. To add class methods, pass the metaclass (`aClass->isa` instead of `aClass`).

*methodList*

A pointer to an `objc_method_list` data structure containing an array of methods to add to the specified class definition.

#### Discussion

All the methods in the specified method list must be mapped to valid selectors before they can be added to the class. You can use the [sel\\_registerName](#) (page 29) function to perform this operation.

After you call `class_addMethods`, the class definition contains the pointer to the method list data structure that you passed in. You cannot release the memory occupied by this data structure until you have removed the methods from the class definition using the [class\\_removeMethods](#) (page 16) function.

### **class\_createInstance**

Creates an instance of a class, allocating memory for the class in the default malloc memory zone.

```
id class_createInstance(Class theClass, unsigned additionalByteCount);
```

#### Parameters

*theClass*

A pointer to an [objc\\_class](#) (page 31) data structure. Pass the class definition that you wish to allocate an instance of.

*additionalByteCount*

An integer indicating the number of extra bytes to allocate. The additional bytes can be used to store additional instance variables beyond those defined in the class definition.

**Return Value**

A pointer to an [objc\\_object](#) (page 37) data structure describing the object instance.

### **class\_createInstanceFromZone**

Creates an instance of a class, allocating memory for the class in a specific malloc memory zone.

```
id class_createInstanceFromZone(Class theClass, unsigned additionalByteCount, void *zone);
```

**Parameters**

*theClass*

A pointer to an [objc\\_class](#) (page 31) data structure. Pass the class definition that you wish to allocate an instance of.

*additionalByteCount*

An integer indicating the number of extra bytes to allocate. The additional bytes can be used to store additional instance variables beyond those defined in the class definition.

*zone*

A pointer to a malloc zone data structure.

**Return Value**

A pointer to an [objc\\_object](#) (page 37) data structure describing the object instance.

### **class\_getClassMethod**

Returns a pointer to the data structure describing the specified class method.

```
Method class_getClassMethod(Class aClass, SEL aSelector);
```

**Parameters**

*aClass*

A pointer to a class definition. Pass the class that contains the method you want to retrieve.

*aSelector*

A pointer of type [SEL](#) (page 34). Pass the selector of the method you want to retrieve.

**Return Value**

A pointer to an [objc\\_method](#) (page 34) data structure. NULL if the specified class does not contain a class method with the specified selector.

**Discussion**

Returns NULL if the specified class does not contain a class method for the specified selector.

### **class\_getInstanceMethod**

Returns a pointer to the data structure describing the specified instance method.

```
Method class_getInstanceMethod(Class aClass, SEL aSelector);
```

#### Parameters

*aClass*

A pointer to an [objc\\_class](#) (page 31) data structure. Pass the class definition whose method you wish to retrieve.

*aSelector*

A pointer of type [SEL](#) (page 34). Pass the selector of the method you want to retrieve.

#### Return Value

A pointer to an [objc\\_method](#) (page 34) data structure. NULL if the specified class does not contain an instance method with the specified selector.

#### Discussion

Returns NULL if the specified class does not contain an instance method for the specified selector.

### **class\_getInstanceVariable**

Obtains information about the instance variables defined for a particular class.

```
Ivar class_getInstanceVariable(Class aClass, const char* aVariableName);
```

#### Parameters

*aClass*

A pointer to an [objc\\_class](#) (page 31) data structure. Pass the class definition whose instance variable you wish to obtain.

*aVariableName*

A pointer to a C string. Pass the name of the instance variable definition to obtain.

#### Return Value

A pointer to an [objc\\_ivar](#) (page 32) data structure containing information about the instance variable specified by *aVariableName*.

#### Discussion

You can use this function to obtain the definition of a single instance variable.

### **class\_getVersion**

Returns the version number of a class definition.

```
int class_getVersion(Class theClass);
```

#### Parameters

*theClass*

A pointer to an [objc\\_class](#) (page 31) data structure. Pass the class definition for which you wish to obtain the version.

#### Return Value

An integer indicating the version number of the class definition.

**Discussion**

You can use the version number of the class definition to provide versioning of the interface that your class represents to other classes. This is especially useful for object serialization (that is, archiving of the object in a flattened form), where it is important to recognize changes to the layout of the instance variables in different class-definition versions.

Classes derived from the Foundation framework `NSObject` class can obtain the class-definition version number using the `getVersion` class method, which is implemented using the `class_getVersion` function.

**class\_nextMethodList**

Returns one of the method lists contained in the specified class definition.

```
struct objc_method_list* class_nextMethodList(Class theClass, void** iterator);
```

**Parameters**

*theClass*

A pointer to an `objc_class` (page 31) data structure. Pass the class definition whose method list you wish to obtain.

*iterator*

On input, a pointer that points to an opaque value. Pass a pointer to a variable with 0 as its value to return the first method list of the class. On output, points to an iteration value, which can be passed to the next call to this function to get the next method list.

**Discussion**

You can obtain all the methods in a class definition by calling the `class_nextMethodList` function repeatedly, passing the same variable to `iterator`.

Listing 1 shows how to use the `class_nextMethodList` function to obtain all the method definitions from a class.

**Listing 1**      Obtaining class method definitions

```
void *iterator = 0;
struct objc_method_list *methodList;

//
// Each call to class_nextMethodList returns one methodList
//
methodList = class_nextMethodList( classObject, &iterator )
while( methodList != NULL )
{
    // ...do something with the method list here...
    methodList = class_nextMethodList( classObject, &iterator );
}
```

**class\_poseAs**

Globally override a specified class definition with a different class definition.

```
Class class_poseAs(Class imposter, Class original);
```

### Parameters

*imposter*

A pointer to an [objc\\_class](#) (page 31) data structure. Pass the class definition that replaces the original. The imposter class definition must be an immediate subclass of the original (that is, `(imposter->super_class == original)` must be a true statement), and the imposter must not contain instance variables (`imposter->ivars` must be NULL.)

*original*

A pointer to an [objc\\_class](#) (page 31) data structure. Pass the class definition you wish to override.

### Discussion

You can use this function to globally override the definition of a class with an immediate subclass. Whenever a request is made of the original class, the imposter class is substituted as the receiver.

Generally, this function is called by the `poseAs:` method of the `NSObject` class, so you should never need to call it yourself.

## class\_removeMethods

Removes a previously added list of methods from a class definition.

```
void class_removeMethods(Class aClass, struct objc_method_list* methodList)
```

### Parameters

*aClass*

A pointer to an [objc\\_class](#) (page 31) data structure. To remove instance methods, pass the class definition from which you wish to remove the methods. To remove class methods, pass the metaclass definition (`aClass->isa` instead of `aClass`).

*methodList*

A pointer to an `objc_method_list` data structure containing an array of methods to remove from the specified class definition. This pointer must be identical to a pointer passed to `class_addMethods` earlier.

### Discussion

You can use this function to remove a list of methods that you have previously added to a class definition using the [class\\_addMethods](#) (page 12) function.

## class\_setVersion

Sets the version number of a class definition.

```
void class_setVersion(Class theClass, int version);
```

### Parameters

*theClass*

A pointer to an [objc\\_class](#) (page 31) data structure. Pass the class definition for which you wish to set the version.

*version*

An integer. Pass the new version number of the class definition.

**Discussion**

You can use the version number of the class definition to provide versioning of the interface that your class represents to other classes. This is especially useful for object serialization (that is, archiving of the object in a flattened form), where it is important to recognize changes to the layout of the instance variables in different class-definition versions.

Classes derived from the Foundation framework `NSObject` class can set the class-definition version number using the `setVersion:` class method, which is implemented using the `class_setVersion` function.

**marg\_free**

Macro that releases an argument list.

```
marg_free(margs)
```

**Parameters**

*margs*

A pointer of type `marg_list` (page 34). Pass the argument list to release.

**marg\_getRef**

Macro that returns a pointer to an argument in an argument list.

```
marg_getRef(margs, offset, type)
```

**Parameters**

*margs*

A pointer of type `marg_list` (page 34).

*offset*

A long integer value. Pass the byte offset to the argument in the list whose pointer you wish to obtain.

*type*

A type name. Pass the type of the argument located at `offset`.

**Return Value**

A pointer to the argument specified by the `offset` parameter.

**Discussion**

You can use this macro to manipulate any sort of `int` or pointer parameter. If you want to handle floats and structs, you should use `NSInvocation` instead.

**marg\_getValue**

Macro that returns the value of an argument in an argument list.

```
marg_getValue(margs, offset, type)
```

**Parameters**

*margs*

A pointer of type `marg_list` (page 34).

*offset*

A long integer value. Pass the byte offset to the argument in the list whose value you wish to obtain.

*type*

A type name. Pass the type of the argument located at `offset`.

#### **Return Value**

The value of the argument specified by the `offset` parameter.

#### **Discussion**

You can use this macro to manipulate any sort of `int` or pointer parameter. If you want to handle floats and structs, you should use `NSInvocation` instead.

### **marg\_malloc**

Macro that allocates an argument list.

```
marg_malloc(margs, method)
```

#### **Parameters**

*margs*

A pointer of type `marg_list` (page 34). Pass the variable that contains the argument list pointer.

*method*

A pointer to an `objc_method` (page 34) data structure. Pass the method for which the argument list is allocated.

#### **Discussion**

You can use this macro to manipulate any sort of `int` or pointer parameter. If you want to handle floats and structs, you should use `NSInvocation` instead.

### **marg\_setValue**

Macro that sets the value of an argument in an argument list.

```
marg_setValue(margs, offset, type, value)
```

#### **Parameters**

*margs*

A pointer of type `marg_list` (page 34).

*offset*

A long integer value. Pass the byte offset to the argument in the list whose pointer you wish to obtain.

*type*

A type name. Pass the type of the argument located at `offset`.

*value*

A value. Pass the new value for the argument.

#### **Discussion**

You can use this macro to manipulate any sort of `int` or pointer parameter. If you want to handle floats and structs, you should use `NSInvocation` instead.

### **method\_getArgumentInfo**

Returns information about one of a method's arguments.

```
unsigned method_getArgumentInfo(Method method, int argIndex, const char** type,  
int* offset);
```

#### Parameters

*method*

A pointer to an [objc\\_method](#) (page 34) data structure. Pass the method in question.

*argIndex*

An integer specifying an argument. Pass the zero-based index of the argument in question.

*type*

On output, a pointer to a C string containing the type encoding for the argument.

*offset*

On output, a pointer to an integer indicating the location of the argument within the memory allocated for the method implementation. The offset is from the start of the implementation memory to the location of the method.

#### Return Value

Zero-based index of the argument that comes before the argument identified by `argIndex`.

#### Discussion

### **method\_getNumberOfArguments**

Returns the number of arguments accepted by a method.

```
unsigned method_getNumberOfArguments(Method method);
```

#### Parameters

*method*

A pointer to a [objc\\_method](#) (page 34) data structure. Pass the method in question.

#### Return Value

An integer containing the number of arguments accepted by the given method.

### **method\_getSizeOfArguments**

Returns the total size of the stack frame occupied by a method's arguments.

```
unsigned method_getSizeOfArguments(Method method);
```

#### Parameters

*method*

A pointer to a [objc\\_method](#) (page 34) data structure. Pass the method in question.

#### Return Value

An integer containing the size of the section of the stack frame occupied by the given method's arguments.

### **objc\_addClass**

Registers a class definition with the Objective-C runtime.

```
void objc_addClass(Class myClass);
```

### Parameters

*myClass*

A pointer to an `objc_class` (page 31) data structure. Pass the class definition you wish to register.

### Discussion

Be sure that you have already set up the class definition correctly, with appropriate links to its metaclass definition and to the definition of its superclass. Listing 2 demonstrates how to properly create and add a class to the Objective-C runtime. Once a class is registered, you can get a pointer to the `objc_class` (page 31) data structure of this class using the Foundation framework function `NSClassFromString`.

For reasons of thread safety, you cannot remove a registered class from the Objective-C runtime.

### Listing 2 Creating an Objective-C class definition

```
#import <objc/objc-runtime.h>

BOOL CreateClassDefinition( const char * name,
                           const char * superclassName )
{
    struct objc_class * meta_class;
    struct objc_class * super_class;
    struct objc_class * new_class;
    struct objc_class * root_class;

    // Ensure that the superclass exists and that someone
    // hasn't already implemented a class with the same name
    //
    super_class = (struct objc_class *)objc_lookUpClass (superclassName);
    if (super_class == nil)
    {
        return NO;
    }

    if (objc_lookUpClass (name) != nil)
    {
        return NO;
    }

    // Find the root class
    //
    root_class = super_class;
    while( root_class->super_class != nil )
    {
        root_class = root_class->super_class;
    }

    // Allocate space for the class and its metaclass
    //
    new_class = calloc( 2, sizeof(struct objc_class) );
    meta_class = &new_class[1];

    // setup class
    new_class->isa      = meta_class;
    new_class->info     = CLS_CLASS;
    meta_class->info    = CLS_META;
```

```

// Create a copy of the class name.
// For efficiency, we have the metaclass and the class itself
// to share this copy of the name, but this is not a requirement
// imposed by the runtime.
//
new_class->name = malloc (strlen (name) + 1);
strcpy ((char*)new_class->name, name);
meta_class->name = new_class->name;

// Allocate empty method lists.
// We can add methods later.
//
new_class->methodLists = calloc( 1, sizeof(struct objc_method_list *) );
*new_class->methodLists = -1;
meta_class->methodLists = calloc( 1, sizeof(struct objc_method_list *) );
*meta_class->methodLists = -1;

// Connect the class definition to the class hierarchy:
// Connect the class to the superclass.
// Connect the metaclass to the metaclass of the superclass.
// Connect the metaclass of the metaclass to the metaclass of the root
class.
//
new_class->super_class = super_class;
meta_class->super_class = super_class->isa;
meta_class->isa = (void *)root_class->isa;

// Set the sizes of the class and the metaclass.
//
new_class->instance_size = super_class->instance_size;
meta_class->instance_size = meta_class->super_class->instance_size;

// Finally, register the class with the runtime.
//
objc_addClass( new_class );
return YES;
}

```

## objc\_getClass

Returns a pointer to the class definition of the specified class.

```
id objc_getClass(const char *aClassName)
```

### Parameters

*aClassName*

A C string. Pass the name of the class to look up.

### Return Value

An `id` pointing to the Class object for the named class, or `nil` if the class is not registered with the Objective-C runtime.

### Discussion

If the specified class is not registered with the Objective-C runtime, `objc_getClass` returns `nil`.

`objc_getClass` is different from `objc_lookupClass` (page 23) in that if the class is not registered, `objc_getClass` calls the class handler callback and then checks a second time to see whether the class is registered. `objc_lookupClass` (page 23) does not call the class handler callback.

### Special Considerations

Earlier implementations of this function (prior to Mac OS X v10.0) terminate the program if the class does not exist.

## objc\_getClassList

Obtains the list of registered class definitions.

```
int objc_getClassList(Class *buffer, int bufferLen);
```

### Parameters

*buffer*

An array of `Class` values. On output, each `Class` value points to one class definition, up to either `bufferLen` or the total number of registered classes, whichever is less. You can pass `NULL` to obtain the total number of registered class definitions without actually retrieving any class definitions.

*bufferLen*

An integer value. Pass the number of pointers for which you have allocated space in `buffer`. On return, this function fills in only this number of elements. If this number is less than the number of registered classes, this function returns an arbitrary subset of the registered classes.

### Return Value

An integer value indicating the total number of registered classes.

### Discussion

The Objective-C runtime library automatically registers all the classes defined in your source code. You can create class definitions at runtime and register them with the `objc_addClass` function.

Listing 3 demonstrates how to use this function to retrieve all the class definitions that have been registered with the Objective-C runtime in the current process.

#### Listing 3 Using objc\_getClassList

```
int numClasses;
Class * classes = NULL;

classes = NULL;
numClasses = objc_getClassList(NULL, 0);

if( numClasses > 0 )
{
    classes = malloc( sizeof(Class) * numClasses );
    (void) objc_getClassList( classes, numClasses );
    free(classes);
}
```

### Special Considerations

Class objects you get from this function cannot be assumed to be classes that inherit from `NSObject`, so you cannot safely call any methods on such classes without detecting that the method is implemented first.

**objc\_getMetaClass**

Returns a pointer to the metaclass definition of the specified class.

```
id objc_getMetaClass(const char *aClassName)
```

**Parameters**

*aClassName*

A C string. Pass the name of the class to look up.

**Return Value**

An `id` pointing to the `Class` object for the metaclass of the named class, or `nil` if the class is not registered with the Objective-C runtime.

**Discussion**

If the specified class is not registered with the Objective-C runtime, this function returns `nil`.

If the definition for the named class is not registered, this function calls the class handler callback and then checks a second time to see if the class is registered. However, every class definition must have a valid metaclass definition, and so the metaclass definition is always returned, whether it's valid or not.

**objc\_lookUpClass**

Returns a pointer to the class definition of the specified class.

```
id objc_lookUpClass(const char *aClassName)
```

**Parameters**

*aClassName*

A C string. Pass the name of the class to look up.

**Return Value**

An `id` pointing to the `Class` object for the named class, or `nil` if the class is not registered with the Objective-C runtime.

**Discussion**

If the specified class is not registered with the Objective-C runtime, this function returns `nil`.

[objc\\_getClass](#) (page 21) is different from this function in that if the class is not registered, [objc\\_getClass](#) (page 21) calls the class handler callback and then checks a second time to see whether the class is registered. This function does not call the class handler callback.

**objc\_msgSend**

Sends a message with a simple return value to an instance of a class.

```
id objc_msgSend(id theReceiver, SEL theSelector, ...)
```

**Parameters**

*theReceiver*

A pointer that points to the instance of the class that is to receive the message.

*theSelector*

The selector of the method that handles the message.

...

A variable argument list containing the arguments to the method.

**Return Value**

The return value of the method.

**Discussion**

When it encounters a method call, the compiler generates a call to one of the functions `objc_msgSend`, `objc_msgSend_stret`, `objc_msgSendSuper`, or `objc_msgSendSuper_stret`. Messages sent to an object's superclass (using the `super` keyword) are sent using `objc_msgSendSuper`; other messages are sent using `objc_msgSend`. Methods that have data structures as return values are sent using `objc_msgSendSuper_stret` and `objc_msgSend_stret`.

**objc\_msgSendSuper**

Sends a message with a simple return value to the superclass of an instance of a class.

```
id objc_msgSendSuper(struct objc_super* superContext, SEL theSelector, ...)
```

**Parameters**

*superContext*

A pointer to an `objc_super` (page 37) data structure. Pass values identifying the context the message was sent to, including the instance of the class that is to receive the message and the superclass at which to start searching for the method implementation.

*theSelector*

A pointer of type `SEL` (page 34). Pass the selector of the method that will handle the message.

...

A variable argument list containing the arguments to the method.

**Return Value**

The return value of the method.

**Discussion**

When it encounters a method call, the compiler generates a call to one of the functions `objc_msgSend`, `objc_msgSend_stret`, `objc_msgSendSuper`, or `objc_msgSendSuper_stret`. Messages sent to an object's superclass (using the `super` keyword) are sent using `objc_msgSendSuper`; other messages are sent using `objc_msgSend`. Methods that have data structures as return values are sent using `objc_msgSendSuper_stret` and `objc_msgSend_stret`.

**objc\_msgSendSuper\_stret**

Sends a message with a data-structure return value to the superclass of an instance of a class.

```
void objc_msgSendSuper_stret(void* stretAddr, struct objc_super* superContext, SEL theSelector, ...);
```

**Parameters**

*stretAddr*

On input, a pointer that points to the address of a block of memory large enough to contain the return value of the method. On output, contains the return value of the method.

*superContext*

A pointer to an [objc\\_super](#) (page 37) data structure. Pass values identifying the context the message was sent to, including the instance of the class that is to receive the message and the superclass at which to start searching for the method implementation.

*theSelector*

A pointer of type [SEL](#) (page 34). Pass the selector of the method.

...

A variable argument list containing the arguments to the method.

**Discussion**

When it encounters a method call, the compiler generates a call to one of the functions `objc_msgSend`, `objc_msgSend_stret`, `objc_msgSendSuper`, or `objc_msgSendSuper_stret`. Messages sent to an object's superclass (using the `super` keyword) are sent using `objc_msgSendSuper`; other messages are sent using `objc_msgSend`. Methods that have data structures as return values are sent using `objc_msgSendSuper_stret` and `objc_msgSend_stret`.

**objc\_msgSendv**

Given an argument list, send a message with a simple return value.

```
id objc_msgSendv(id self, SEL op, unsigned arg_size, marg_list arg_frame);
```

**Parameters***self*

A pointer to the instance of the class that is to receive the message.

*op*

A pointer of type [SEL](#) (page 34). Pass the selector of the method.

*arg\_size*

The number of elements in *arg\_frame*.

*arg\_frame*

A pointer to an [marg\\_list](#) (page 34). Pass an argument list containing the values for the arguments of the method being invoked.

**Discussion**

This method is used by the `NSInvocation` class.

**Declared In**

`objc-runtime.h`

**objc\_msgSendv\_fpret**

Given an argument list, send a message with a floating point return value.

```
double objc_msgSendv_fpret(id self, SEL op, unsigned arg_size, marg_list arg_frame);
```

**Parameters***self*

A pointer to the instance of the class that is to receive the message.

*op*

A pointer of type [SEL](#) (page 34). Pass the selector of the method.

*arg\_size*

The number of elements in *arg\_frame*.

*arg\_frame*

A pointer to an [marg\\_list](#) (page 34). Pass an argument list containing the values for the arguments of the method being invoked.

#### Discussion

This method is used by the `NSInvocation` class.

On the i386 platform, the ABI for functions returning a floating-point value is incompatible with that for functions returning an integral type. On the i386 platform, therefore, you *must* use `objc_msgSendv_fpret` for functions that for functions returning non-integral type. For `float` or `long double` return types, cast the function to an appropriate function pointer type first.

This function is not used on the PPC or PPC64 platforms.

### objc\_msgSendv\_stret

Given an argument list, send a message with a data-structure return value.

```
void objc_msgSendv_stret(void* stretAddr, id self, SEL op, unsigned arg_size,
marg_list arg_frame);
```

#### Parameters

*stretAddr*

On input, a pointer that points to the address of a block of memory large enough to contain the return value of the method. On output, contains the return value of the method.

*self*

A pointer to the instance of the class that is to receive the message.

*op*

A pointer of type `SEL` (page 34). Pass the selector of the method.

*arg\_frame*

A pointer to an [marg\\_list](#) (page 34). Pass an argument list containing the values for the arguments of the method being invoked.

#### Discussion

This method is used by the `NSInvocation` class.

### objc\_msgSend\_fpret

Sends a message with a floating-point return value to an instance of a class.

```
double objc_msgSend_fpret(id self, SEL op, ...)
```

#### Parameters

*self*

A pointer that points to the instance of the class that is to receive the message.

*op*

The selector of the method that handles the message.

*...*

A variable argument list containing the arguments to the method.

**Discussion**

On the i386 platform, the ABI for functions returning a floating-point value is incompatible with that for functions returning an integral type. On the i386 platform, therefore, you *must* use `objc_msgSend_fpret` for functions that for functions returning non-integral type. For `float` or `long double` return types, cast the function to an appropriate function pointer type first.

This function is not used on the PPC or PPC64 platforms.

**Declared In**

`objc-runtime.h`

**objc\_msgSend\_stret**

Sends a message with a data-structure return value to an instance of a class.

```
void objc_msgSend_stret(void * stretAddr, id theReceiver, SEL theSelector, ...);
```

**Parameters**

*stretAddr*

On input, a pointer that points to a block of memory large enough to contain the return value of the method. On output, contains the return value of the method.

*theReceiver*

A pointer to the instance of the class that is to receive the message.

*theSelector*

A pointer of type [SEL](#) (page 34). Pass the selector of the method that handles the message.

...

A variable argument list containing the arguments to the method.

**Discussion**

When it encounters a method call, the compiler generates a call to one of the functions `objc_msgSend`, `objc_msgSend_stret`, `objc_msgSendSuper`, or `objc_msgSendSuper_stret`. Messages sent to an object's superclass (using the `super` keyword) are sent using `objc_msgSendSuper`; other messages are sent using `objc_msgSend`. Methods that have data structures as return values are sent using `objc_msgSendSuper_stret` and `objc_msgSend_stret`.

**objc\_setClassHandler**

Sets a custom error-handling callback function called from [objc\\_getClass](#) (page 21) and [objc\\_getMetaClass](#) (page 23) when the desired class is not found.

```
void objc_setClassHandler(int (*callback)(const char *));
```

**Parameters**

*callback*

A function pointer. Pass the callback conforming to the prototype described in [ClassHandlerCallback](#) (page 30).

**Discussion**

If the [objc\\_getClass](#) (page 21) function is unable to find a specified class, it calls the class handler callback that you specify with this function.

The callback can use the [objc\\_addClass](#) (page 19) function to register the class definition.

After the callback is invoked, [objc\\_getClass](#) (page 21) and [objc\\_getMetaClass](#) (page 23) search for the named class, again.

### **object\_getInstanceVariable**

Obtains the value of an instance variable of a class instance.

```
Ivar object_getInstanceVariable(id object, const char *name, void **value);
```

#### **Parameters**

*object*

A pointer to an instance of a class. Pass the object containing the instance variable whose value you wish to obtain.

*name*

A C string. Pass the name of the instance variable whose value you wish to obtain.

*value*

A pointer to a pointer to a value. On output, contains a pointer to the value of the instance variable.

#### **Return Value**

A pointer to the [objc\\_ivar](#) (page 32) data structure that defines the type and name of the instance variable specified by *name*.

### **object\_setInstanceVariable**

Changes the value of an instance variable of a class instance.

```
Ivar object_setInstanceVariable(id object, const char *name, void *value);
```

#### **Parameters**

*object*

A pointer to an instance of a class. Pass the object containing the instance variable whose value you wish to modify.

*name*

A C string. Pass the name of the instance variable whose value you wish to modify.

*value*

A pointer to a value. Pass a pointer to the new value for the instance variable.

#### **Return Value**

A pointer to the [objc\\_ivar](#) (page 32) data structure that defines the type and name of the instance variable specified by *name*.

### **sel\_getName**

Returns the name of the method specified by the selector.

```
const char* sel_getName(SEL aSelector)
```

#### **Parameters**

*aSelector*

A pointer of type [SEL](#) (page 34). Pass the selector whose name you wish to determine.

**Return Value**

A C string indicating the name of the selector.

**Discussion**

This function is one of two ways to retrieve the name of a selector. You can also cast a selector pointer directly to a C string pointer (`const char *`), but you cannot use an arbitrary C string as a selector because selectors are C strings that are indexed by pointer. See [SEL](#) (page 34) for more information.

**sel\_getUid**

Registers a method name with the Objective-C runtime system.

```
SEL sel_getUid(const char *str);
```

**Parameters**

*str*

A pointer to a C string. Pass the name of the method you wish to register.

**Return Value**

A pointer of type [SEL](#) (page 34) specifying the selector for the named method.

**Discussion**

The implementation of this method is identical to the implementation of [sel\\_registerName](#) (page 29).

**Version Notes**

Prior to Mac OS X version 10.0, this method tried to find the selector mapped to the given name and returned `NULL` if the selector was not found. This was changed for safety, because it was observed that many of the callers of this function did not check the return value for `NULL`. You can still use the [sel\\_isMapped](#) (page 29) function to determine whether a method name is mapped to a selector.

**sel\_isMapped**

Indicates whether a selector is registered.

```
BOOL sel_isMapped(SEL aSelector)
```

**Parameters**

*aSelector*

A pointer of type [SEL](#) (page 34). Pass the selector whose validity you wish to determine.

**Return Value**

A Boolean value indicating whether the selector is valid (YES) or not (NO).

**Discussion**

You can use this function to determine whether a given address is a valid selector (that is, one that has been registered). Use of this function may require you to subvert the compiler's type checking by casting the address value to `SEL`. See [SEL](#) (page 34) for more information.

**sel\_registerName**

Registers a method with the Objective-C runtime system, maps the method name to a selector, and returns the selector value.

```
SEL sel_registerName(const char *str);
```

**Parameters**

*str*

A pointer to a C string. Pass the name of the method you wish to register.

**Return Value**

A pointer of type [SEL](#) (page 34) specifying the selector for the named method.

**Discussion**

You must register a method name with the Objective-C runtime system to obtain the method's selector before you can add the method to a class definition. If the method name has already been registered, this function simply returns the selector.

## Callbacks

### ClassHandlerCallback

Handles creation and registration of a class when [objc\\_getClass](#) (page 21) or [objc\\_getMetaClass](#) (page 23) cannot otherwise find the class.

```
int (*ClassHandlerCallback)(const char* className);
```

If you name your function `MyClassHandlerCallback`, you would declare it like this:

```
int MyClassHandlerCallback(const char*
className);
```

**Parameters**

*className*

A C string indicating the name of the class that could not be found.

**Return Value**

Return 1 if your handler registered a class; otherwise, return 0.

## Data Types

### Class-Definition Data Structures

---

The compiler records Objective-C class definitions as [objc\\_class](#) (page 31) data structures. You can create additional [objc\\_class](#) (page 31) data structures at runtime and add them to the class hierarchy using the Objective-C runtime functions. The compiler places the types and names of the instance variables in the [objc\\_ivar\\_list](#) (page 33) data structure, and the types and names of methods in the [objc\\_method\\_list](#) (page 35) data structure. Using the current Objective-C support library, you can add methods to any class at runtime, but you cannot add instance variables. The structures described in this section are defined in `/usr/include/objc/objc-class.h`.

- [objc\\_class](#) (page 31) defines an Objective-C class.

- [objc\\_ivar](#) (page 32) specifies the name, type and location of one instance variable.
- [objc\\_ivar\\_list](#) (page 33) specifies the instance variables of each instance of a class.
- [IMP](#) (page 33) pointer to the start of a method implementation.
- [marg\\_list](#) (page 34) reference to an argument list.
- [SEL](#) (page 34) represents a method selector.
- [objc\\_method](#) (page 34) represents a method in a class definition.
- [objc\\_method\\_list](#) (page 35) Contains an array of method definitions.
- [objc\\_cache](#) (page 35) pointers to recently used methods.
- [objc\\_protocol\\_list](#) (page 36) represents a list of formal protocols.

## objc\_class

Defines an Objective-C class.

```
struct objc_class
{
    struct objc_class* isa;
    struct objc_class* super_class;
    const char* name;
    long version;
    long info;
    long instance_size;
    struct objc_ivar_list* ivars;
    struct objc_method_list** methodLists;
    struct objc_cache* cache;
    struct objc_protocol_list* protocols;
};
```

### Fields

`isa`

Points to the metaclass of this class. If this class is a metaclass, the `isa` field points to the root metaclass (usually the metaclass for `NSObject`, but also possibly the metaclass for `NSProxy` or a root class of your own—a root class is any class that does not inherit from another class). This also means that the `isa` field for the metaclass of the root class points to itself.

`super_class`

Points to the `objc_class` data structure for the superclass of this class, or `NULL` if this is a root class.

`name`

Points to a C string containing the name of the class.

`version`

An integer indicating the version number of the class, which you can modify at runtime using the [class\\_setVersion](#) (page 16) function. The compiler initially defines the `version` field as 0.

`info`

Contains a set of bit flags used by the Objective-C runtime functions. You can manipulate them using the following masks:

- The `CLS_CLASS` (0x1L) flag indicates that this definition represents a class, which contains instance methods and variable definitions that are allocated for each new instance of the class.
- The `CLS_META` (0x2L) flag indicates that this class definition represents a metaclass, which contains the list of methods that are not specific to any one instance of the class (class methods).
- The `CLS_INITIALIZED` (0x4L) flag indicates that the runtime has initialized this class. This flag should be set only by the `objc_addClass` (page 19) function.
- The `CLS_POSING` (0x8L) flag indicates that this class is posing as another class.
- The `CLS_MAPPED` (0x10L) flag is used internally by the Objective-C runtime.
- The `CLS_FLUSH_CACHE` (0x20L) flag is used internally by the Objective-C runtime.
- The `CLS_GROW_CACHE` (0x40L) flag is used internally by the Objective-C runtime.
- The `CLS_NEED_BIND` (0x80L) flag is used internally by the Objective-C runtime.
- The `CLS_METHOD_ARRAY` (0x100L) flag indicates that the `methodLists` field is an array of pointers to `objc_method_list` (page 35) data structures rather than a pointer to a single `objc_method_list` (page 35) data structure.

`instance_size`

An integer indicating the size of the instance variables used by this class. This value includes the value of the `instance_size` field of the superclass.

`ivars`

A pointer to an `objc_ivar_list` (page 33) data structure describing the instance variables that are allocated for each instance of this class. This pointer may be `NULL`, in which case this class has no instance variables.

`methodLists`

If the `CLS_METHOD_ARRAY` flag is set, this field is an array of `objc_method_list` (page 35) data structures that collectively specify all the instance methods that can be sent to objects that are instances of this class. If the `CLS_METHOD_ARRAY` flag is not set, this field is a pointer to a single `objc_method_list` (page 35) data structure. If this class is a metaclass definition, this field specifies the class methods of the class.

`cache`

A pointer to an `objc_cache` (page 35) method cache data structure.

`protocols`

A pointer to a `objc_protocol_list` (page 36) data structure. This is a list of the formal protocols this class claims to implement.

**Discussion**

The compiler generates two `objc_class` (page 31) data structures for each class defined in your source code: one for the class definition and one for the metaclass definition. You can create class definitions at runtime by creating these data structures and calling the `objc_addClass` (page 19) function.

**`objc_ivar`**

Specifies the name, type, and location of one instance variable.

```
typedef struct objc_ivar *Ivar;

struct objc_ivar
{
    char *ivar_name;
    char *ivar_type;
    int ivar_offset;
};
```

**Fields**`ivar_name`

A pointer to a C string containing the name of the instance variable.

`ivar_type`

A pointer to a C string containing the type encoding of the variable. See “Type Encodings” for valid type encodings for instance variables.

`ivar_offset`

An integer indicating the location of this instance variable within the memory allocated for an instance of the class containing this variable. The offset is from the start of the instance memory to the location of this variable.

**Discussion**

The `objc_ivar_list` (page 33) data structure contains an array of `objc_ivar` elements, each of which indicates the name, type, and location of one instance variable.

**objc\_ivar\_list**

Specifies the instance variables of each instance of a class.

```
struct objc_ivar_list
{
    int ivar_count;
    struct objc_ivar ivar_list[1];
};
```

**Fields**`ivar_count`

An integer specifying a the number of elements in the `ivar_list` array.

`ivar_list`

An array of `objc_ivar` (page 32) data structures.

**Discussion**

This data structure contains an array with of `objc_ivar` (page 32) elements, each of which indicates the name, type, and location of one instance variable.

**IMP**

A pointer to the start of a method implementation.

```
id (*IMP)(id, SEL, ...)
```

### Discussion

This data type is a pointer to the start of the function that implements the method. This function uses standard C calling conventions as implemented for the current CPU architecture. The first argument is a pointer to `self` (that is, the memory for the particular instance of this class, or, for a class method, a pointer to the metaclass). The second argument is the method selector. The method arguments follow.

### marg\_list

A reference to an argument list.

```
typedef void * marg_list;
```

### Discussion

This data type is a reference to a list of method arguments. Use it with the functions described in “[Forwarding Messages](#)” (page 8).

### SEL

Represents a method selector.

```
typedef struct objc_selector;
```

### Discussion

Method selectors are used to represent the name of a method at runtime. A method selector is a C string that has been registered (or “mapped”) with the Objective-C runtime. Selectors generated by the compiler are automatically mapped by the runtime when the class is loaded.

You can add new selectors at runtime and retrieve existing selectors using the function [sel\\_registerName](#) (page 29).

When using selectors, you must use the value returned from [sel\\_registerName](#) (page 29) or the Objective-C compiler directive `@selector()`. You cannot simply cast a C string to SEL.

### objc\_method

Represents a method in a class definition.

```
struct objc_method
{
    SEL method_name;
    char * method_types;
    IMP method_imp;
};
```

### Fields

`method_name`

A pointer of type [SEL](#) (page 34). Points to the method selector that uniquely identifies the name of this method.

`method_types`

A pointer to a C string. This string contains the type encodings for the method's argument. See "Type Encodings" for information on valid encoding formats.

`method_imp`

A pointer to the start of the method implementation. In a class definition that represents a formal protocol, this field is `NULL`.

### Special Considerations

The compiler generates the method type encodings in a format that includes information on the size of the stack and the size occupied by the arguments. These numbers appear after each encoding in the `method_types` string. However, because the compiler historically generates them incorrectly, and because they differ depending on the CPU type, the runtime ignores them if they are present. These numbers are not required by the Objective-C runtime in Mac OS X v10.0 or later.

## `objc_method_list`

Contains an array of method definitions.

```
struct objc_method_list
{
    struct objc_method_list *obsolete;
    int method_count;
    struct objc_method method_list[1];
}
```

### Fields

`obsolete`

Reserved for future use.

`method_count`

An integer specifying the number of methods in the method list array.

`method_list`

An array of [objc\\_method](#) (page 34) data structures.

## `objc_cache`

Performance optimization for method calls. Contains pointers to recently used methods.

```
struct objc_cache
{
    unsigned int mask;
    unsigned int occupied;
    Method buckets[1];
};
```

### Fields

`mask`

An integer specifying the total number of allocated cache buckets (minus one). During method lookup, the Objective-C runtime uses this field to determine the index at which to begin a linear search of the `buckets` array. A pointer to a method's selector is masked against this field using a logical AND operation (`index = (mask & selector)`). This serves as a simple hashing algorithm.

occupied

An integer specifying the total number of occupied cache buckets.

buckets

An array of pointers to [objc\\_method](#) (page 34) data structures. This array may contain no more than `mask + 1` items. Note that pointers may be `NULL`, indicating that the cache bucket is unoccupied, and occupied buckets may not be contiguous. This array may grow over time.

### Discussion

To limit the need to perform linear searches of method lists for the definitions of frequently accessed methods—an operation that can considerably slow down method lookup—the Objective-C runtime functions store pointers to the definitions of the most recently called method of the class in an `objc_cache` data structure.

## objc\_protocol\_list

Represents a list of formal protocols.

```
struct objc_protocol_list
{
    struct objc_protocol_list *next;
    int count;
    Protocol *list[1];
};
```

### Fields

next

A pointer to another `objc_protocol_list` data structure.

count

The number of protocols in this list.

list

An array of pointers to [objc\\_class](#) (page 31) data structures that represent protocols.

### Discussion

A formal protocol is a class definition that declares a set of methods, which a class must implement. Such a class definition contains no instance variables. A class definition may promise to implement any number of formal protocols.

## Instance Data Types

---

These are the data types that represent objects, classes, and superclasses (they are defined in `/usr/include/objc/objc-class.h`):

- [id](#) (page 36) pointer to an instance of a class.
- [objc\\_object](#) (page 37) represents an instance of a class.
- [objc\\_super](#) (page 37) specifies the superclass of an instance.

### id

A pointer to an instance of a class.

```
typedef struct objc_object *id;
```

**Discussion**

An `id` is a pointer to an [objc\\_object](#) (page 37) data structure.

**objc\_object**

Represents an instance of a class.

```
struct objc_object
{
    struct objc_class *isa;
    /* ...variable length data containing instance variable values... */
};
```

**Fields**

`isa`

A pointer to the class definition of which this object is an instance.

**Discussion**

When you create an instance of a particular class, the allocated memory contains an `objc_object` data structure, which is directly followed by the data for the instance variables of the class.

The `alloc` and `allocWithZone:` methods of the Foundation framework class `NSObject` use the functions [class\\_createInstance](#) (page 12) and [class\\_createInstanceFromZone](#) (page 13) to create `objc_object` data structures.

**objc\_super**

Specifies the superclass of an instance.

```
struct objc_super
{
    id receiver;
    Class class;
};
```

**Fields**

`receiver`

A pointer of type `id` (page 36). Specifies an instance of a class.

`class`

A pointer to an [objc\\_class](#) (page 31) data structure. Specifies the particular superclass of the instance to message.

**Discussion**

The compiler generates an `objc_super` data structure when it encounters the `super` keyword as the receiver of a message. It specifies the class definition of the particular superclass that should be messaged.

## Macros

---

These macros improve the readability of Objective-C code. They are defined in `/usr/include/objc/objc.h`.

## **YES**

Defines YES as 1.

```
#define YES (BOOL)1
```

## **NO**

Defines NO as 0.

```
#define NO (BOOL)0
```

# Document Revision History

---

This table describes the changes to *Objective-C 1 Runtime Reference*.

Date	Notes
2009-01-06	Added link to current document.
2007-03-28	Legacy version of the document originally released as Objective-C Runtime Reference.

**REVISION HISTORY**

Document Revision History

# Index

---

## C

---

ClassHandlerCallback **callback** 30  
class\_addMethods **function** 12  
class\_createInstance **function** 12  
class\_createInstanceFromZone **function** 13  
class\_getClassMethod **function** 13  
class\_getInstanceMethod **function** 13  
class\_getInstanceVariable **function** 14  
class\_getVersion **function** 14  
class\_nextMethodList **function** 15  
class\_poseAs **function** 15  
class\_removeMethods **function** 16  
class\_setVersion **function** 16

## I

---

id **data type** 36  
IMP **data type** 33

## M

---

marg\_free **function** 17  
marg\_getRef **function** 17  
marg\_getValue **function** 17  
marg\_list **data type** 34  
marg\_malloc **function** 18  
marg\_setValue **function** 18  
method\_getArgumentInfo **function** 18  
method\_getNumberOfArguments **function** 19  
method\_getSizeOfArguments **function** 19

## N

---

NO **data type** 38

## O

---

objc\_addClass **function** 19  
objc\_cache **structure** 35  
objc\_class **structure** 31  
objc\_getClass **function** 21  
objc\_getClassList **function** 22  
objc\_getMetaClass **function** 23  
objc\_ivar **structure** 32  
objc\_ivar\_list **structure** 33  
objc\_lookupClass **function** 23  
objc\_method **structure** 34  
objc\_method\_list **structure** 35  
objc\_msgSend **function** 23  
objc\_msgSendSuper **function** 24  
objc\_msgSendSuper\_stret **function** 24  
objc\_msgSendv **function** 25  
objc\_msgSendv\_fpret **function** 25  
objc\_msgSendv\_stret **function** 26  
objc\_msgSend\_fpret **function** 26  
objc\_msgSend\_stret **function** 27  
objc\_object **structure** 37  
objc\_protocol\_list **structure** 36  
objc\_setClassHandler **function** 27  
objc\_super **structure** 37  
object\_getInstanceVariable **function** 28  
object\_setInstanceVariable **function** 28

## S

---

SEL **data type** 34  
@selector() **directive** 34  
sel\_getName **function** 28  
sel\_getUid **function** 29  
sel\_isMapped **function** 29  
sel\_registerName **function** 29

## Y

---

YES data type [38](#)