

## NSAllocateObject(), NSDeallocateObject()

**SUMMARY** Create and destroy objects

**DECLARED IN** foundation/NSObject.h

### SYNOPSIS

```
id *NSAllocateObject(Class class, unsigned extraBytes, NXZone *zone)  
void NSDeallocateObject(id <NSObject> anObject)
```

**DESCRIPTION** **NSAllocateObject** allocates and returns a pointer to an instance of *class*, created in the specified zone (or in the default zone, if *zone* is NULL). The *extraBytes* argument (usually zero) states the number of extra bytes required for indexed instance variables. Returns **nil** on failure.

**NSDeallocateObject** deallocates *anObject*, which must have been allocated using **NSAllocateObject()**.

**RETURN** **NSAllocateObject** returns a pointer to an instance of *class*, or **nil** upon failure.

**NSDeallocateObject** returns **void**.

**SEE ALSO** NSCopyObject()

## NSAssert, NSAssertn, NSCAssert, NSCAssertn, NSParameterAssert, NSCParameterAssert

**SUMMARY** Assertion macros

DECLARED IN     foundation/NSExceptions.h

SYNOPSIS **NSAssert**(*condition*, NSString \**description*)

**NSAssert1**(*condition*, NSString \**description*, *arg1* )

**NSAssert2**(*condition*, NSString \**description*, *arg1*, *arg2*)

**NSAssert3**(*condition*, NSString \**description*, *arg1*, *arg2*, *arg3*)

**NSAssert4**(*condition*, NSString \**description*, *arg1*, *arg2*, *arg3*, *arg4*)

**NSAssert5**(*condition*, NSString \**description*, *arg1*, *arg2*, *arg3*, *arg4*, *arg5*)

**NSCAssert**(*condition*, NSString \**description*)

**NSCAssert1**(*condition*, NSString \**description*, *arg1* )

**NSCAssert2**(*condition*, NSString \**description*, *arg1*, *arg2*)

**NSCAssert3**(*condition*, NSString \**description*, *arg1*, *arg2*, *arg3*)

**NSCAssert4**(*condition*, NSString \**description*, *arg1*, *arg2*, *arg3*, *arg4*)

**NSCAssert5**(*condition*, NSString \**description*, *arg1*, *arg2*, *arg3*, *arg4*, *arg5*)

**NSParameterAssert**(*condition*)

**NSCParameterAssert**(*condition*)

**DESCRIPTION**     Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class `NSAssertionHandler`. When invoked, an assertion handler prints an error message that includes method and class (or function name). It then raises an exception of type `NSInternalInconsistencyException`.

An assortment of macros evaluate the condition and serve as a front end to the assertion handler. These macros fall into two types: those for use within Objective-C methods (**NSAssertn()**), and those for use within C functions (**NSCAssertn()**). **NSAssert()** and **NSCAssert()** take no arguments other than the condition and the format string. The other macros take the number of format-string arguments as indicated by *n*.

*condition* must be an expression that evaluates to true or false. *description* is a **printf()**-style format string that describes the failure condition. Each *arg* is an argument to be inserted, in place, into the *description*.

**NSParameterAssert()** and **NSCParameterAssert()** are assertion macros that validate parameters, one within Objective-C methods and the other within C functions. Simply provide the parameter as the *condition* argument.

The macro evaluates the parameter and, if it is false, it logs an error message which includes the parameter and raises an exception

Assertions are compiled into code only if the preprocessor macro **DEBUG** is defined.

**RETURN** All macros return **void**.

**SEE ALSO** **NSRaise()**, **NSRaisev()**, **NSLog()**, **NSLogv()**

## **NSClassFromString(), NSStringFromClass**

**SUMMARY** Obtain a class by name, or the name of a class

**DECLARED IN** foundation/NSObjCRuntime.h

### **SYNOPSIS**

Class **NSClassFromString**(NSString \**aClassName*)  
NSString \***NSStringFromClass**(Class *aClass*)

**DESCRIPTION** **NSClassFromString** returns the class object named by *aClassName*, or **nil** if none by this name is currently loaded.

**NSStringFromClass** returns an NSString containing the name of *aClass*.

## **NSCopyObject()**

**SUMMARY** Copy NSObjects

**DECLARED IN** foundation/NSObject.h

## SYNOPSIS

NSObject \***NSCopyObject**(NSObject \**anObject*, unsigned *extraBytes*, NSZone \**zone*)

**DESCRIPTION**     Creates and returns a new object that's an exact copy of *anObject*, created in the specified zone (or in the default zone, if *zone* is NULL). The *extraBytes* argument (usually zero) states the number of extra bytes required for indexed instance variables. Returns **nil** on failure.

**RETURN**     Returns a pointer to a new NSObject that is an exact copy of *anObject*.

**SEE ALSO**    **NSAllocateObject()**, **NSDeallocateObject()**

## NSCreateZone()

**SUMMARY**    Creates a new zone

**DECLARED IN**    foundation/NSZone.h

## SYNOPSIS

NSZone \***NSCreateZone**(unsigned *startSize*, unsigned *granularity*, BOOL *canFree*)

**DESCRIPTION**     Creates and returns a pointer to a new zone of *startSize* bytes, which will grow and shrink by *granularity* bytes. If *canFree* is zero, the allocator will never free memory, and **malloc()** will be fast.

**RETURN**     Returns a pointer to a new NSZone.

**SEE ALSO**    **NSDefaultMallocZone()**, **NSRecycleZone()**, **NSSetZoneName()**

## **NSDefaultMallocZone()**

**SUMMARY** Returns the default zone

**DECLARED IN** foundation/NSZone.h

### **SYNOPSIS**

NSZone \***NSDefaultMallocZone**(void)

**DESCRIPTION** Returns the default zone, which is created automatically at startup. This is the zone used by the standard C function **malloc()**.

**RETURN** Returns a pointer to the default zone.

**SEE ALSO** **NSCreateZone()**

## **NSGetUncaughtExceptionHandler(), NSSetUncaughtExceptionHandler()**

**SUMMARY** Change the top level error handler.

**DECLARED IN** foundation/NSException.h

### **SYNOPSIS**

NSUncaughtExceptionHandler \***NSGetUncaughtExceptionHandler**(void)  
void **NSSetUncaughtExceptionHandler**(NSUncaughtExceptionHandler \**handler*)

**DESCRIPTION** **NSGetUncaughtExceptionHandler** returns a pointer to the function serving as the top-level error handler. This handler will process exceptions raised outside of any exception-handling domain.

**NSSetUncaughtExceptionHandler** sets the top-level error-handling function to *handler*. If *handler* is NULL or

this function is never invoked, the default top-level handler is used.

**RETURN** **NSGetUncaughtExceptionHandler** returns a pointer to the top-level error handler.

**NSSetUncaughtExceptionHandler** returns **void**.

## **NSIncrementExtraRefCount(), NSDecrementExtraRefCountWasZero()**

**SUMMARY** Modify object reference counts

**DECLARED IN** foundation/NSObject.h

### **SYNOPSIS**

```
void NSIncrementExtraRefCount(id anObject)  
BOOL NSDecrementExtraRefCountWasZero(id anObject)
```

**DESCRIPTION** These functions modify the <sup>a</sup>extra reference<sup>o</sup> count of an object. Newly created objects have only one actual reference, so that a single **release** message results in the object being deallocated. Extra references are those beyond the single original reference, and are usually created by sending the object a **retain** message. Your code should generally not use these functions unless it's overriding the **retain** or **release** methods.

**RETURN** **NSDecrementExtraRefCountWasZero()** returns NO if *anObject* had an extra reference count. If *anObject* didn't have an extra referenct count, it returns YES, indicating that the object should be deallocated (with **dealloc**).

## **NSIntersectionRange(), NSUnionRange()**

**SUMMARY** Combine ranges

**DECLARED IN**     foundation/NSRange.h

**SYNOPSIS**

NSRange **NSIntersectionRange**(NSRange *range1*, NSRange *range2*)

NSRange **NSUnionRange**(NSRange *range1*, NSRange *range2*)

**DESCRIPTION**     **NSIntersectionRange()** returns a range describing the intersection of *range1* and *range2*—that is, a range containing the indices that exist in both ranges. If the returned range's **length** field is zero, then the two ranges don't intersect, and the value of the **location** field is undefined.

**NSUnionRange()** returns a range covering all indices in and between *range1* and *range2*. If one range is completely contained in the other, the returned range is equal to the larger range.

**RETURN**     Each function returns the resulting combined range.

## **NSLocationInRange(), NSMaxRange()**

**SUMMARY** Check positions in ranges

**DECLARED IN**     foundation/NSRange.h

**SYNOPSIS**

BOOL **NSLocationInRange**(unsigned *index*, NSRange *aRange*)

unsigned **NSMaxRange**(NSRange *aRange*)

**DESCRIPTION**     **NSLocationInRange()** returns YES if the given *index* lies within *aRange*—that is, if it's greater than or equal to *aRange.location* and less than *aRange.location* plus *aRange.length*.

**NSMaxRange()** returns the location plus the length of *aRange*. This is the index for the item just *past* the end of

the range, not the last item in the range. Note that **NSMaxRange()** can easily overflow if *aRange.length* is a large value.

## **NSLog(), NSLogv()**

**SUMMARY** Log an error message to **stderr**.

**DECLARED IN** foundation/NSUtilities.h

### **SYNOPSIS**

```
extern void NSLog(NSString *format, ...)
extern void NSLogv(NSString *format, va_list args)
```

**DESCRIPTION** **NSLogv()** logs an error message to **stderr**. The message consists of a timestamp and the process ID prefixed to the string you pass in. You compose this string with a format string and a variable number of arguments. **NSLog()** simply passes along a variable number of arguments to **NSLogv()**.

*format* is a **printf()**-style format string. Following this, in **NSLog()**, are one or more arguments to be inserted into the string.

**RETURN** Both functions return **void**.

**SEE ALSO** **NSRaise(), NSRaisev()**

## **NSRaise(), NSRaisev()**

**SUMMARY** Raise an exception.



**DECLARED IN**     foundation/NSExceptions.h

**SYNOPSIS**

```
extern void NSRaise(unsigned int exceptionCode, NSString *format, ...)
extern void NSRaisev(unsigned int exceptionCode, NSString *format, va_list args)
```

**DESCRIPTION**     **NSRaisev()** logs an error message and then raises an exception (**NX\_RAISE()**). **NSRaise()** simply passes along a variable number of arguments to **NSRaisev()**.

*exceptionCode* is a value of type **NSException** that indicates the basis of the error. *format* is a **printf()**-style format string. Following this string, in **NSRaise()**, are one or more arguments to be inserted into the string. Use **NSRaise()** in place of **NX\_RAISE()**.

**RETURN**     Both functions return **void**.

**SEE ALSO**     **NSAssert()**, **NSLog()**, **NSLogv()**

## **NSRecycleZone()**

**SUMMARY**     Frees memory in a zone

**DECLARED IN**     foundation/NSZone.h

**SYNOPSIS**

```
void NSRecycleZone(NSZone *zone)
void NSZoneFree(NSZone *zone, void *pointer)
```

**DESCRIPTION**     **NSRecycleZone** frees *zone* after adding any of its pointers still in use to the default zone. (This strategy prevents retained objects from being inadvertently destroyed.)

**NSZoneFree** returns the memory indicated by *pointer* to *zone*. The standard C function **free()** does the same, but spends time finding which zone the memory belongs to.

**RETURN** Both functions return **void**.

**SEE ALSO** **NSCreateZone()**, **NSZoneMalloc()**

## **NSSelectorFromString(), NSStringFromSelector()**

**SUMMARY** Obtain a selector by name, or the name of a selector

**DECLARED IN** foundation/NSObjCRuntime.h

### **SYNOPSIS**

```
SEL NSSelectorFromString(NSString *aSelectorName)
NSString *NSStringFromSelector(SEL aSelector)
```

**DESCRIPTION** **NSSelectorFromString** returns the selector named by *aSelectorName*, or zero if none by this name exists.

**NSStringFromSelector** returns an NSString containing the name of *aSelector*.

## **NSSetZoneName(), NSZoneName()**

**SUMMARY** Work with zone names

**DECLARED IN** foundation/NSZone.h

### **SYNOPSIS**

```
void NSSetZoneName(NSZone *zone, NSString *name)
NSString *NSZoneName(NSZone *zone)
```

**DESCRIPTION**     **NSSetZoneName** sets the specified *zone*'s name to *name*, which can aid in debugging.

**NSZoneName** returns the name of the specified *zone* as an NSString.

## **NSShouldRetainWithZone()**

**SUMMARY** Decide whether to retain an object

**DECLARED IN**     foundation/NSObject.h

### **SYNOPSIS**

```
BOOL NSShouldRetainWithZone(NSObject *anObject, NSZone *requestedZone)
```

**DESCRIPTION**     Returns YES if *requestedZone* is NULL, the default zone, or the zone in which *anObject* was allocated. This function is typically called from inside an NSObject's **copyWithZone:** method, when deciding whether to retain *anObject* as opposed to making a copy of it.

**RETURN**     Returns YES if *anObject* should be retained with *requestedZone*.

## **NSStringFromRange()**

**SUMMARY** Get a string representation of a range

**DECLARED IN**     foundation/NSRange.h

#### SYNOPSIS

NSString \***NSStringFromRange**(NSRange *range*)

**DESCRIPTION** Returns a string representation of the specified *range*.

**RETURN** Returns a string of the form:  $\{location = a; length = b\}^o$ , where *a* and *b* are non-negative integers.

#### **NSUserName(), NSHomeDirectory(), NSHomeDirectoryForUser()**

**SUMMARY** Get information about a user

**DECLARED IN** foundation/NSPathUtilities.h

#### SYNOPSIS

NSString \***NSUserName**(void)

NSString \***NSHomeDirectory**(void)

NSString \***NSHomeDirectoryForUser**(NSString \* *userName*)

**DESCRIPTION** **NSUserName** returns the name of the current user.

**NSHomeDirectory** returns a path to the current user's home directory.

**NSHomeDirectoryForUser** returns a path to the home directory for the user specified by *userName*.

#### **NSZoneCalloc(), NSZoneMalloc(), NSZoneRealloc()**

**SUMMARY** Allocate memory in a zone

**DECLARED IN**     foundation/NSZone.h

**SYNOPSIS**

```
void *NSZoneCalloc(NSZone *zone, unsigned numElems, unsigned byteSize)  
void *NSZoneMalloc(NSZone *zone, unsigned size)  
void *NSZoneRealloc(NSZone *zone, void *ptr, unsigned size)
```

**DESCRIPTION**     **NSZoneCalloc** allocates enough memory from *zone* for *numElems* elements, each with a size *numBytes* bytes, and returns a pointer to the allocated memory. The memory is initialized with zeros.

**NSZoneMalloc** allocates *size* bytes in *zone*, and returns a pointer to the allocated memory.

**NSZoneRealloc** changes the size of the block of memory pointed to by *ptr* to *size* bytes. It may allocate new memory to replace the old, in which case it moves the contents of the old memory block to the new block, up to a maximum of *size* bytes. *ptr* may be NULL.

**RETURN**     All three functions return a pointer to the newly-allocated block of memory, or **nil** if the operation was unable to allocate the requested memory.

**SEE ALSO**   **NSDefaultMallocZone()**, **NSRecycleZone()**, **NSZoneFree()**

## **NSZoneFromPtr**

**SUMMARY**     Get the zone for a given block of memory

**DECLARED IN**     foundation/NSZone.h

**SYNOPSIS**

```
NSZone *NSZoneFromPtr(void *pointer)
```

**DESCRIPTION** Returns the zone for the block of memory indicated by *pointer*, or NULL if the block was not allocated from a zone. The pointer must be one that was returned by a prior call to an allocation function.

**RETURN** Returns the zone for the indicated block of memory, or NULL if the block was not allocated from a zone.

**SEE ALSO** **NSZoneCalloc()**, **NSZoneMalloc()**, **NSZoneRealloc()**