# Functions

---

## NSAllHashTableObjects()

**DECLARED IN** Foundation/NSHashTable.h

**SYNOPSIS** NSArray ***NSAllHashTableObjects**(NSHashTable **table*)

**DESCRIPTION** Returns an array object containing all the elements of *table*. This function should be called only when the table elements are objects, not when they're any other data type.

---

## NSAllMapTableKeys(), NSAllMapTableValues()

**DECLARED IN** Foundation/NSMapTable.h

**SYNOPSIS** NSArray ***NSAllMapTableKeys**(NSMapTable **table*)
NSArray ***NSAllMapTableValues**(NSMapTable **table*)

**DESCRIPTION** **NSAllMapTableKeys()** Returns an array object containing all the keys in *table*. This function should be called only when the table keys are objects, not when they're any other type of pointer.

**NSAllMapTableValues()** Returns an array object containing all the values in *table*. This function should be called only when the table values are objects, not when they're any other type of pointer.

**SEE ALSO** **NSMapMember(), NSMapGet(), NSEnumerateMapTable(), NSNextMapEnumeratorPair()**

## 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()**

---

## NSAllocateMemoryPages(), NSDeallocateMemoryPages()

**DECLARED IN** Foundation/NSZone.h

**SYNOPSIS** void ***NSAllocateMemoryPages**(unsigned *byteCount*)
void **NSDeallocateMemoryPages**(void **pointer*, unsigned *byteCount*)

**DESCRIPTION** **NSAllocateMemoryPages()** allocates the integral number of pages whose total size is closest to, but not less than, *byteCount*, with the pages guaranteed to be zero-filled. **NSDeallocateMemoryPages()** deallocates memory that was allocated with **NSAllocateMemoryPages()**.

**SEE ALSO** **NSCopyMemoryPages()**

---

## NSAssert, NSAssert*n*, NSCAssert, NSCAssert*n*, 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 a 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 (**NSAssert*n*()**), and those for use within C functions (**NSCAssert*n*()**). **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*.

---

## NSCompareHashTables()

**DECLARED IN** Foundation/NSHashTable.h

**SYNOPSIS** BOOL **NSCompareHashTables**(NSHashTable *_table1_, NSHashTable *_table2_)

**DESCRIPTION** Returns YES if the two hash tables are equal—that is, if each element of *table1* is in *table2*, and the two tables are the same size.

---

## NSCompareMapTables()

**DECLARED IN** Foundation/NSMapTable.h

**SYNOPSIS** BOOL **NSCompareMapTables**(NSMapTable *_table1_, NSMapTable *_table2_)

**DESCRIPTION** Returns YES if each key of *table1* is in *table2*, and the two tables are the same size. Note that this function does not compare values, only keys.

## NSContainsRect()

**DECLARED IN** Foundation/NSGeometry.h

**SYNOPSIS** BOOL **NSContainsRect**(NSRect *aRect*, NSRect *bRect*)

**DESCRIPTION** Returns YES if *aRect* completely encloses *bRect*. For this to be true, *bRect* can't be empty and none of its sides can touch any of *aRect*'s.

---

## NSCopyHashTableWithZone()

**DECLARED IN** Foundation/NSHashTable.h

**SYNOPSIS** NSHashTable ***NSCopyHashTableWithZone**(NSHashTable **table*, NSZone **zone*)

**DESCRIPTION** Returns a pointer to a new copy of *table*, created in *zone* and containing copies of *table*'s pointers to data elements. If *zone* is NULL, the default zone is used.

**SEE ALSO** **NSCreateHashTable(), NSCreateHashTableWithZone()**

## NSCopyMapTableWithZone()

**DECLARED IN** Foundation/NSMapTable.h

**SYNOPSIS** NSMapTable *__NSCopyMapTableWithZone__(NSMapTable **table*, NSZone **zone*)

**DESCRIPTION** Returns a pointer to a new copy of *table*, created in *zone* and containing copies of *table*'s key and value pointers. If *zone* is NULL, the default zone is used.

**SEE ALSO** **NSCreateMapTable(), NSCreateMapTableWithZone()**

---

## NSCopyMemoryPages()

**DECLARED IN** Foundation/NSZone.h

**SYNOPSIS** void **NSCopyMemoryPages**(const void **source*, void **destination*,unsigned *byteCount*)

**DESCRIPTION** Copies (or copies-on-write) *byteCount* bytes from *source* to *destination*.

**SEE ALSO** **NSAllocateMemoryPages(), NSDeallocateMemoryPages()**

## 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()**

---

## NSCountHashTable()

**DECLARED IN** Foundation/NSHashTable.h

**SYNOPSIS** unsigned **NSCountHashTable**(NSHashTable **table*)

**DESCRIPTION** Returns the number of elements in *table*.

## NSCountMapTable()

**DECLARED IN** Foundation/NSMapTable.h

**SYNOPSIS** unsigned **NSCountMapTable**(NSMapTable *_table_)

**DESCRIPTION** Returns the number of key/value pairs in *table*.

---

## NSCreateHashTable(), NSCreateHashTableWithZone()

**DECLARED IN** Foundation/NSHashTable.h

**SYNOPSIS** NSHashTable ***NSCreateHashTable**(NSHashTableCallBacks *callBacks*, unsigned *capacity*)
NSHashTable ***NSCreateHashTableWithZone**(NSHashTableCallBacks *callBacks*, unsigned *capacity*, NSZone **zone*)

**DESCRIPTION** **NSCreateHashTable()** creates and returns a pointer to an NSHashTable in the default zone. The table's size is dependent on (but generally not equal to) *capacity*. If *capacity* is 0, a small hash table is created. The NSHashTableCallBacks structure *callBacks* has five pointers to functions (documented under "Types and Constants"), with the following defaults: pointer hashing, if **hash()** is NULL; pointer equality, if **isEqual()** is NULL; no call-back upon adding an element, if **retain()** is NULL; no call-back upon removing an element, if **release()** is NULL; and a function returning a pointer's hexadecimal value as a string, if **describe()** is NULL. The hashing function must be defined such that if two data elements are equal, as defined by the comparison function, the values produced by hashing on these elements must also be equal. Also, data elements must remain invariant if the value of the hashing function depends on them; for example, if the hashing function operates directly on the characters of a string, that string can't change.

**NSCreateHashTableWithZone()** is like **NSCreateHashTable()**, but creates the hash table in *zone* instead of in the default zone. (If *zone* is NULL, the default zone is used.)

**SEE ALSO** **NSCopyHashTableWithZone()**

---

## NSCreateMapTable(), NSCreateMapTableWithZone()

**DECLARED IN** Foundation/NSMapTable.h

**SYNOPSIS** NSMapTable *__NSCreateMapTable__(NSMapTableKeyCallBacks *keyCallBacks*, NSMapTableValueCallBacks *valueCallBacks*, unsigned *capacity*)
NSMapTable *__NSCreateMapTableWithZone__(NSMapTableKeyCallBacks *keyCallBacks*, NSMapTableValueCallBacks *valueCallBacks*, unsigned *capacity*, NSZone **zone*)

**DESCRIPTION** **NSCreateMapTable()** creates, and returns a pointer to, an NSMapTable in the default zone; the table's size is dependent on (but generally not equal to) *capacity*. If *capacity* is 0, a small map table is created. The NSMapTableKeyCallBacks arguments are structures (documented under "Types and Constants") that are very similar to the call-back structure used by **NSCreateHashTable()**; in fact, they have the same defaults as documented for that function.

**NSCreateMapTableWithZone()** is like **NSCreateMapTable()**, but creates the map table in *zone* instead of in the default zone. (If *zone* is NULL, the default zone is used.)

**SEE ALSO** **NSCopyMapTableWithZone()**

## 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()**

---

## NSDecimalAdd()

**DECLARED IN** Foundation/NSDecimal.h

**SYNOPSIS** NSCalculationError **NSDecimalAdd**(NSDecimal **result*, const NSDecimal **leftOperand*, const NSDecimal **rightOperand*, NSRoundingMode *roundingMode*)

**DESCRIPTION** Adds *leftOperand* to *rightOperand*, and stores the sum in *result*.

An NSDecimal can represent a number with up to 38 significant digits. If a number is more precise than that, it must be rounded off. *roundingMode* determines how to round it off. There are four possible rounding modes:

- NSRoundDown. The number rounds down.
- NSRoundUp. The number rounds up.
- NSRoundPlain. The number rounds to the closest 38-digit approximation. If the number is halfway between two positive numbers, it round up; if it's halfway between two negative numbers, it rounds down.
- NSRoundBankers. The number rounds to the closest 38-digit approximation. If it is caught halfway between two possibilities, it rounds to the one whose last digit is even. In practice, this means that, over the long run, numbers will be rounded up as often as they are rounded down; there will be no systematic bias.

The return value indicates whether any machine limitations were encountered in the addition. If none were encountered, the function returns NSCalculationNoError. Otherwise it may return one of the following values: NSCalculationLossOfPrecision, NSCalculationOverflow or NSCalculationUnderflow. For descriptions of all these error conditions, see **exceptionDuringOperation:error:leftOperand:rightOperand** in the protocol specification for NSDecimalNumberBehaviors.

---

## NSDecimalCompact()

**DECLARED IN** Foundation/NSDecimal.h

**SYNOPSIS** void **NSDecimalCompact**(NSDecimal *_number_)

**DESCRIPTION** Formats *number* so that calculations using it will take up as little memory as possible. All the **NSDecimal...** arithmetic functions expect compact NSDecimal arguments.

## NSDecimalCompare()

**DECLARED IN** Foundation/NSDecimal.h

**SYNOPSIS** NSComparisonResult **NSDecimalCompare**(const NSDecimal ***leftOperand*, const NSDecimal **rightOperand*)

**DESCRIPTION** Compares *leftOperand* to *rightOperand*, with three possible return values.

- If *leftOperand* is bigger, the function returns NSOrderedDescending.
- If *rightOperand* is bigger, the the function returns NSOrdered Ascending.
- If the two operands are equal, the function returns NSOrderedSame.

## NSDecimalDivide()

**DECLARED IN** Foundation/NSDecimal.h

**SYNOPSIS** NSCalculationError **NSDecimalDivide**(NSDecimal ***result*, const NSDecimal ***leftOperand*, const NSDecimal **rightOperand*, NSRoundingMode *roundingMode*)

**DESCRIPTION** Divides *leftOperand* by *rightOperand*, and stores the quotient, possibly rounded off according to *roundingMode*, in *result*. If *rightOperand* is 0, returns NSDivideByZero.

For explanations of the other possible return values, and of all the possible *roundingMode*'s, see **NSDecimalAdd**(), above.

Note that this function can't precisely represent a non-decimal fraction like 1/3.

## NSDecimalMultiply()

**DECLARED IN** Foundation/NSDecimal.h

**SYNOPSIS** NSCalculationError **NSDecimalMultiply**(NSDecimal ***result*, const NSDecimal ***leftOperand*, const NSDecimal ***rightOperand*, NSRoundingMode *roundingMode*)

**DESCRIPTION** Multiplies *rightOperand* by *leftOperand*, and stores the product, possibly rounded off according to *roundingMode*, in *result*.

For explanations of the possible return values and *roundingMode*'s, see **NSDecimalAdd**(), above.

## NSDecimalMultiplyByPowerOf10()

**DECLARED IN** Foundation/NSDecimal.h

**SYNOPSIS** NSCalculationError NSDecimalMultiplyByPowerOf10(NSDecimal ***result*, const NSDecimal ***number*, short *power*, NSRoundingMode *roundingMode*)

**DESCRIPTION** Multiplies *number* by $10^{power}$, and stores the product, possibly rounded off according to *roundingMode*, in *result*.

For explanations of the possible return values and *roundingMode*'s, see **NSDecimalAdd**(), above.

## NSDecimalNormalize()

**DECLARED IN** Foundation/NSDecimal.h

**SYNOPSIS** NSCalculationError **NSDecimalNormalize**(NSDecimal ***number1*, NSDecimal ***number2*, NSRoundingMode *roundingMode*)

**DESCRIPTION** An NSDecimal is represented in memory as a mantissa and an exponent, expressing the value mantissa **x** $10^{exponent}$. A number can have many NSDecimal representations; for example, the following are all valid NSDecimal representations for the number 100:

| Mantissa | Exponent |
|---|---|
| 100 | 0 |
| 10 | 1 |
| 1 | 2 |

**NSDecimalNormalize** formats number1 and number2 so that they have equal exponents. This format makes addition and subtraction very convenient. Both **NSDecimalAdd**() and **NSDecimalSubtract**() call **NSDecimalNormalize**(). You may want to use it if you write more complicated addition or subtraction routines.

For explanations of the function's possible return values, see **NSDecimalAdd**(), above.

## NSDecimalPower()

**DECLARED IN** Foundation/NSDecimal.h

**SYNOPSIS** NSCalculationError **NSDecimalPower**(NSDecimal **result*, const NSDecimal **number*, unsigned *power*, NSRoundingMode *roundingMode*)

**DESCRIPTION** Raises *number* to *power*, and stores the result, possibly rounded off according to *roundingMode*, in *result*.

For explanations of the possible return values and *roundingMode*'s, see **NSDecimalAdd**(), above.

---

## NSDecimalRound()

**DECLARED IN** Foundation/NSDecimal.h

**SYNOPSIS** void **NSDecimalRound**(NSDecimal **result*, const NSDecimal **number*, int *scale*, NSRoundingMode *roundingMode*)

**DESCRIPTION** Rounds *number* off according to the parameters *scale* and rounding *mode*, and stores the result in *result*.

*scale* specifies the number of digits result can have after its decimal point. *roundingMode* specifies the way that number is rounded off. There are four possible values for *roundingMode*: NSRoundDown, NSRoundUp, NSRoundPlain, and NSRoundBankers. For thorough discussions of *scale* and *roundingMode*, see the **scale** and **roundingMode** in the protocol specification for NSDecimalNumberBehaviors.

## NSDecimalString()

**DECLARED IN** Foundation/NSDecimal.h

**SYNOPSIS** NSString ***NSDecimalString**(const NSDecimal **decimal*, NSDictionary **locale*)

**DESCRIPTION** Returns a string representation of *decimal*. *locale* determines whether the NSDecimalSeparator should be a period (as in the United States) or a comma (as in France).

---

## NSDecimalSubtract()

**DECLARED IN** Foundation/NSDecimal.h

**SYNOPSIS** NSCalculationError **NSDecimalSubtract**(NSDecimal **result*, const NSDecimal **leftOperand*, const NSDecimal **rightOperand*, NSRoundingMode *roundingMode*)

**DESCRIPTION** Subtracts *rightOperand* from *leftOperand*, and stores the difference, possibly rounded off according to *roundingMode*, in *result*.

For explanations of the possible return values and *roundingMode*'s, see NSDecimalAdd(), above.

## 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()**

---

## NSDivideRect()

**DECLARED IN** Foundation/NSGeometry.h

**SYNOPSIS** void **NSDivideRect**(NSRect *inRect*, NSRect **slice*, NSRect **remainder*, float *amount*, NSRectEdge *edge*)

**DESCRIPTION** Creates two rectangles, *slice* and *remainder*, from *inRect*, by dividing *inRect* with a line that's parallel to one of *inRect*'s sides (namely, the side specified by edge—either NSMinXEdge,

NSMinYEdge, NSMaxXEdge, or NSMaxYEdge). The size of *slice* is determined by *amount*, which measures the distance from *edge*.

SEE ALSO **NSInsetRect(), NSIntegralRect(), NSOffsetRect()**

---

## NSEnumerateHashTable(), NSNextHashEnumeratorItem()

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSHashEnumerator **NSEnumerateHashTable**(NSHashTable ***table*)
void ***NSNextHashEnumeratorItem**(NSHashEnumerator ***enumerator*)

DESCRIPTION Returns an **NSHashEnumerator** structure that will cause successive elements of *table* to be returned each time this enumerator is passed to **NSNextHashEnumeratorItem()**.

**NSNextHashEnumeratorItem()** returns the next element in the table that enumerator is associated with, or NULL if *enumerator* has already iterated over all the elements.

---

## NSEnumerateMapTable(), NSNextMapEnumeratorPair()

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSMapEnumerator **NSEnumerateMapTable**(NSMapTable ***table*)
BOOL **NSNextMapEnumeratorPair**(NSMapEnumerator ***enumerator*, void ****key*, void ****value*)

DESCRIPTION **NSEnumerateMapTable()** returns an **NSMapEnumerator** structure that will cause successive key/value pairs of *table* to be visited each time this enumerator is passed to **NSNextMapEnumeratorPair()**.

**NSNextMapEnumeratorPair()** returns NO if *enumerator* has already iterated over all the elements in the table that *enumerator* is associated with. Otherwise, this function sets *key* and *value* to match the next key/value pair in the table, and returns YES.

SEE ALSO **NSMapMember(), NSMapGet(), NSAllMapTableKeys(), NSAllMapTableValues()**

---

## NSEqualPoints()

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSEqualPoints**(NSPoint *aPoint*, NSPoint *bPoint*)

DESCRIPTION Returns YES if the two points *aPoint* and *bPoint* are identical, and NO otherwise.

---

## NSEqualRanges()

DECLARED IN Foundation/NSRange.h

SYNOPSIS BOOL **NSEqualRanges**(NSRange *range1*, NSRange *range2*)

DESCRIPTION Returns YES if *range1* and *range2* have the same locations and lengths.

## NSEqualRects()

**DECLARED IN** Foundation/NSGeometry.h

**SYNOPSIS** BOOL **NSEqualRects**(NSRect *aRect*, NSRect *bRect*)

**DESCRIPTION** Returns YES if the two rectangles *aRect* and *bRect* are identical, and NOotherwise.

---

## NSEqualSizes()

**DECLARED IN** Foundation/NSGeometry.h

**SYNOPSIS** BOOL **NSEqualSizes**(NSSize *aSize*, NSSize *bSize*)

**DESCRIPTION** Returns YES if the two sizes *aSize* and *bSize* are identical, and NO otherwise.

## NSFreeHashTable(), NSResetHashTable()

**DECLARED IN** Foundation/NSHashTable.h

**SYNOPSIS** void **NSFreeHashTable**(NSHashTable *_table_)
void **NSResetHashTable**(NSHashTable *_table_)

**DESCRIPTION** **NSFreeHashTable()** releases each element of the specified hash table and frees the table itself. **NSResetHashTable()** releases each element but doesn't deallocate the table. This is useful for preserving the table's capacity.

---

## NSFreeMapTable(), NSResetMapTable()

**DECLARED IN** Foundation/NSMapTable.h

**SYNOPSIS** void **NSFreeMapTable**(NSMapTable *_table_)
void **NSResetMapTable**(NSMapTable *_table_)

**DESCRIPTION** **NSFreeMapTable()** releases each key and value of the specified map table and frees the table itself. **NSResetMapTable()** releases each key and value but doesn’t deallocate the table. This is useful for preserving the table’s capacity.

## 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**.

---

## NSHashGet()

**DECLARED IN** Foundation/NSHashTable.h

**SYNOPSIS** void ***NSHashGet**(NSHashTable **table*, const void **pointer*)

**DESCRIPTION** Returns the pointer in the table that matches *pointer* (as defined by the **isEqual()** call-back function). If there is no matching element, the function returns NULL

## NSHashInsert(), NSHashInsertKnownAbsent(), NSHashInsertIfAbsent()

**DECLARED IN** Foundation/NSHashTable.h

**SYNOPSIS** void **NSHashInsert**(NSHashTable *_table_, const void *_pointer_)
void **NSHashInsertKnownAbsent**(NSHashTable *_table_, const void *_pointer_)
void ***NSHashInsertIfAbsent**(NSHashTable *_table_, const void *_pointer_)

**DESCRIPTION** **NSHashInsert()** inserts *pointer*, which must not be NULL, into *table*. If *pointer* matches an item already in the table, the previous pointer is released using the **release()** call-back function that was specified when the table was created.

**NSHashInsertKnownAbsent()** inserts *pointer*, which must not be NULL, into *table*. Unike **NSHashInsert()**, this function raises NSInvalidArgumentException if *table* already includes an element that matches *pointer.*

If *pointer* matches an item already in *table*, **NSHashInsertIfAbsent()** returns the pre-existing pointer; otherwise, it adds *pointer* to the table and returns NULL.

**SEE ALSO** **NSHashRemove()**

---

## NSHashRemove()

**DECLARED IN** Foundation/NSHashTable.h

**SYNOPSIS** void **NSHashRemove**(NSHashTable *_table_, const void *_pointer_)

**DESCRIPTION** If *pointer* matches an item already in *table*, this function releases the pre-existing item.

**SEE ALSO** **NSHashInsert(), NSHashInsertKnownAbsent(), NSHashInsertIfAbsent()**

---

## NSHeight(), NSWidth()

**DECLARED IN** Foundation/NSGeometry.h

**SYNOPSIS** float **NSHeight**(NSRect *aRect*)
float **NSWidth**(NSRect *aRect*)

**DESCRIPTION** NSStringFromHashTable()

**DESCRIPTION** **NSHeight()** returns the height of *aRect*. **NSWidth()** returns the width of *aRect*.

**SEE ALSO** **NSMaxX(), NSMaxY(), NSMidX(), NSMidY(), NSMinX(), NSMinY()**

---

## 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 "extra reference" 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**).

---

## NSInsetRect(), NSIntegralRect(), NSOffsetRect()

**DECLARED IN** Foundation/NSGeometry.h

**SYNOPSIS** NSRect **NSInsetRect**(NSRect *aRect*, float *dX*, float *dY*)
NSRect **NSIntegralRect**(NSRect *aRect*)
NSRect **NSOffsetRect**(NSRect *aRect*, float *dX*, float *dY*)

**DESCRIPTION** **NSInsetRect()** returns a copy of the rectangle *aRect*, altered by moving the two sides that are parallel to the y-axis inwards by *dX*, and the two sides parallel to the x-axis inwards by *dY.*

**NSIntegralRect()** returns a copy of the rectangle *aRect*, expanded outwards just enough to ensure that none of its four defining values (*x*, *y*, *width*, and *height*) have fractional parts. If *aRect*'s width or height is zero or negative, this function returns a rectangle with origin at (0.0, 0.0) and with zero width and height.

**NSOffsetRect**() returns a copy of the rectangle *aRect*, with its location shifted by *dX* along the x-axis and by *dY* along the y-axis.

**SEE ALSO** **NSDivideRect()**

## 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.

---

## NSIsEmptyRect()

**DECLARED IN** Foundation/NSGeometry.h

**SYNOPSIS** BOOL **NSIsEmptyRect**(NSRect *aRect*)

**DESCRIPTION** Returns YES if the rectangle encloses no area at all—that is, if its width or height is zero or negative.

## NSLocationInRange()

**SUMMARY** Check a position in a range

**DECLARED IN** Foundation/NSRange.h

**SYNOPSIS** BOOL **NSLocationInRange(**unsigned *index*, 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.

---

## 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()**

---

## NSMakePoint()

**DECLARED IN** Foundation/NSGeometry.h

**SYNOPSIS** NSPoint **NSMakePoint(**float *x*, float *y***)**

**DESCRIPTION** Creates an NSPoint having the coordinates *x* and *y*.

---

## NSMakeRect()

**DECLARED IN** Foundation/NSGeometry.h

**SYNOPSIS** NSRect **NSMakeRect(**float *x*, float *y*, float *w*, float *h***)**

**DESCRIPTION** Creates an NSRect having the specified origin and size.

## NSMakeRange()

**DECLARED IN** Foundation/NSRange.h

**SYNOPSIS** NSRange **NSMakeRange(**unsigned int *location*, unsigned int *length***)**

**DESCRIPTION** Creates an NSRange having the specified location and length.

## NSMakeSize()

**DECLARED IN** Foundation/NSGeometry.h

**SYNOPSIS** NSSize **NSMakeSize(**float *w*, float *h***)**

**DESCRIPTION** Creates an NSSize having the specified width and height.

## NSMapGet()

**DECLARED IN** Foundation/NSMapTable.h

**SYNOPSIS** void ***NSMapGet**(NSMapTable **table*, const void **key*)

**DESCRIPTION** Returns the value that table maps to key, or NULL if the table doesn't contain key.

**SEE ALSO** **NSMapMember(), NSEnumerateMapTable(), NSNextMapEnumeratorPair(), NSAllMapTableKeys(), NSAllMapTableValues()**

---

## NSMapInsert(), NSMapInsertIfAbsent(), NSMapInsertKnownAbsent()

**DECLARED IN** Foundation/NSMapTable.h

**SYNOPSIS** void **NSMapInsert**(NSMapTable **table*, const void **key*, const void **value*)
void ***NSMapInsertIfAbsent**(NSMapTable **table*, const void **key*,const void **value*)
void **NSMapInsertKnownAbsent**(NSMapTable **table*, const void **key*, const void **value*)

**DESCRIPTION** **NSMapInsert()** inserts *key* and *value* into table. If *key* matches a key already in the table, *value* is retained and the previous value is released, using the retain and release call-back functions that were specified when the table was created. Raises InvalidArgumentException if *key* is equal to the **notAKeyMarker** field of the table's NSMapTableKeyCallBacks structure.

If *key* matches a key already in table, **NSMapInsertIfAbsent()** returns the pre-existing key; otherwise, it adds *key* and *value* to the table and returns NULL. Raises NSInvalidArgumentException if *key* is equal to the **notAKeyMarker** field of the table's NSMapTableKeyCallBacks structure.

**NSMapInsertKnownAbsent()** inserts *key* (which must not be **notAKeyMarker**) and *value* into table. Unlike **NSMapInsert()**, this function raises NSInvalidArgumentException if *table* already includes a key that matches *key.*

**SEE ALSO** **NSMapRemove()**

---

## NSMapMember()

**DECLARED IN** Foundation/NSMapTable.h

**SYNOPSIS** BOOL **NSMapMember**(NSMapTable *_table_, const void *_key_, void **_originalKey_, void **_value_)

**DESCRIPTION** Returns YES if table contains a key equal to *key.* If so, *originalKey* is set to *key*, and *value* is set to the value that the table maps to *key.*

**SEE ALSO** **NSMapGet(), NSEnumerateMapTable(), NSNextMapEnumeratorPair(), NSAllMapTableKeys(), NSAllMapTableValues()**

---

## NSMapRemove()

**DECLARED IN** Foundation/NSMapTable.h

**SYNOPSIS** void **NSMapRemove**(NSMapTable *_table_, const void *_key_)

**DESCRIPTION** If *key* matches a key already in *table*, this function release the pre-existing key and its corresponding value.

**SEE ALSO** **NSMapInsert(), NSMapInsertIfAbsent(), NSMapInsertKnownAbsent()**

---

## NSMaxRange()

**DECLARED IN** Foundation/NSRange.h

**SYNOPSIS** unsigned **NSMaxRange**(NSRange *range*)

**DESCRIPTION** Returns *range*.location + *range*.length—in other words, the number one greater than the maximum value within the range.

---

## NSMaxX(), NSMaxY(), NSMidX(), NSMidY(), NSMinX(), NSMinY()

**DECLARED IN** Foundation/NSGeometry.h

**SYNOPSIS** float **NSMaxX**(NSRect *aRect*)
float **NSMaxY**(NSRect *aRect*)
float **NSMidX**(NSRect *aRect*)
float **NSMidY**(NSRect *aRect*)
float **NSMinX**(NSRect *aRect*)
float **NSMinY**(NSRect *aRect*)

**DESCRIPTION** **NSMaxX()** returns the largest x-coordinate value within *aRect*. **NSMaxY()** returns the largest y-coordinate value within *aRect*.

**NSMidX()** returns the x-coordinate of the rectangle's center point. **NSMidY()** returns the y-coordinate of the rectangle's center point.

**NSMinX()** returns the smallest x-coordinate value within *aRect*. **NSMinY()** returns the smallest y-coordinate value within *aRect* .

**SEE ALSO** **NSWidth(), NSHeight()**

---

## NSMouseInRect()

**DECLARED IN** Foundation/NSGeometry.h

**SYNOPSIS** BOOL **NSMouseInRect**(NSPoint *aPoint*, NSRect *aRect*, BOOL *flipped*)

**DESCRIPTION** Returns YES if the point represented by *aPoint* is located within the rectangle represented by *aRect*. It assumes an unscaled and unrotated coordinate system; the argument *flipped* should be YES if the coordinate system has been flipped so that the positive y-axis extends downward. This function is used to determine whether the hot spot of the cursor lies inside a given rectangle.

## NSPageSize(), NSLogPageSize()

**DECLARED IN** Foundation/NSZone.h

**SYNOPSIS** unsigned **NSPageSize**(void)
unsigned **NSLogPageSize**(void)

**DESCRIPTION** **NSPageSize()** returns the number of bytes in a page. **NSLogPageSize()** returns the binary log of the page size.

**SEE ALSO** **NSRoundDownToMultipleOfPageSize(), NSRoundUpToMultipleOfPageSize()**

---

## NSPointInRect()

**DECLARED IN** Foundation/NSGeometry.h

**SYNOPSIS** BOOL **NSPointInRect**(NSPoint *aPoint*, NSRect *aRect*)

**DESCRIPTION** Performs the same test as **NSMouseInRect()**, but assumes a flipped coordinate system.

## 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()**

---

## NSRealMemoryAvailable()

**DECLARED IN** Foundation/NSZone.h

**SYNOPSIS** unsigned **NSRealMemoryAvailable**(void)

**DESCRIPTION** **NSRealMemoryAvailable()** returns the number of bytes available in RAM.

## 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()**

---

## NSRoundDownToMultipleOfPageSize(), NSRoundUpToMultipleOfPageSize()

**DECLARED IN** Foundation/NSZone.h

**SYNOPSIS** unsigned **NSRoundDownToMultipleOfPageSize**(unsigned *byteCount*)
unsigned **NSRoundUpToMultipleOfPageSize**(unsigned *byteCount*)

**DESCRIPTION** **NSRoundDownToMultipleOfPageSize()** returns the multiple of the page size that is closest to, but not greater than, *byteCount*. **NSRoundUpToMultipleOfPageSize()** returns the multiple of the page size that is closest to, but not less than, *byteCount*.

**SEE ALSO** **NSPageSize(), NSLogPageSize()**

---

## 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*.

## NSStringFromHashTable()

**DECLARED IN** Foundation/NSHashTable.h

**SYNOPSIS** NSString ***NSStringFromHashTable**(NSHashTable **table*)

**DESCRIPTION** Returns a string describing the hash table's contents. The function iterates over the table's elements, and for each one appends the string returned by the **describe()** call-back function. If NULL was specified for the call-back function, the hexadecimal value of each pointer is added to the string.

---

## NSStringFromMapTable()

**DECLARED IN** Foundation/NSMapTable.h

**SYNOPSIS** NSString ***NSStringFromMapTable**(NSMapTable **table*)

**DESCRIPTION** Returns a string describing the map table's contents. The function iterates over the table's key/value pairs, and for each one appends the string "*a* = *b*;\n", where *a* and *b* are the key and value strings returned by the corresponding **describe()** call-back functions. If NULL was specified for the call-back function, a and b are the key and value pointers, expressed as hexadecimal numbers.

## NSStringFromPoint()

**SUMMARY** Get a string representation of a point

**DECLARED IN** Foundation/NSGeometry.h

**SYNOPSIS** NSString ***NSStringFromPoint**(NSPoint *aPoint*)

**DESCRIPTION** **NSStringFromPoint()** returns a string of the form "{x=*a*; y=*b*}", where *a* and *b* are the x- and y-coordinates of *aPoint*.

---

## NSStringFromRange()

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

**DECLARED IN** Foundation/NSRange.h

**SYNOPSIS** NSString ***NSStringFromRange**(NSRange *aRange*)

**DESCRIPTION** **NSStringFromRange()** returns a string of the form: "{location = *a*; length = *b*}", where *a* and *b* are non-negative integers representing *aRange*.

## NSStringFromRect()

**SUMMARY** Get a string representation of a rect

**DECLARED IN** Foundation/NSGeometry.h

**SYNOPSIS** NSString ***NSStringFromRect**(NSRect *aRect*)

**DESCRIPTION** **NSStringFromRect()** returns a string of the form "{x=*a*; y=*b*; width=*c*; height=*d*}", where *a*, *b*, *c*, and *d* are the x- and y-coordinates and the width and height, respectively, of *aRect*.

---

## NSStringFromSize()

**SUMMARY** Get a string representation of a size

**DECLARED IN** Foundation/NSGeometry.h

**SYNOPSIS** NSString ***NSStringFromSize**(NSSize *aSize*)

**DESCRIPTION** **NSStringFromSize()** returns a string of the form "{width=*a*; height=*b*}", where *a* and *b* are the width and height of *aSize*.

## NSUnionRect(), NSIntersectionRect()

**DECLARED IN** Foundation/NSGeometry.h

**SYNOPSIS**
NSRect **NSUnionRect**(NSRect *aRect*, NSRect *bRect*)
NSRect **NSIntersectionRect**(NSRect *aRect*, NSRect *bRect*)

**DESCRIPTION** **NSUnionRect()** returns the smallest rectangle that completely encloses both *aRect* and *bRect*. If one of the rectangles has zero (or negative) width or height, a copy of the other rectangle is returned; but if both have zero (or negative) width or height, the returned rectangle has its origin at (0.0, 0.0) and has zero width and height.

**NSIntersectionRect()** returns the graphic intersection of *aRect* and *bRect*. If the two rectangles don't overlap, the returned rectangle has its origin at (0.0, 0.0) and zero width and height. (This includes situations where the intersection is a point or a line segment.)

---

## 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()**

## NSZoneFree()

**DECLARED IN** Foundation/NSZone.h

**SYNOPSIS** **NSZoneFree**(NSZone *_zone_, void *_pointer_)

**DESCRIPTION** Returns memory to the zone from which it was allocated. The standard C function **free()** does the same, but spends time finding which zone the memory belongs to.

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

---

## NSZoneFromPointer

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

**DECLARED IN** Foundation/NSZone.h

**SYNOPSIS** NSZone ***NSZoneFromPointer**(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()**