Foundation Kit Functions

Memory Allocation Functions

Get the virtual memory page size:

unsigned **NSPageSize(**void**)** Returns the number of bytes in a page.

unsigned **NSLogPageSize(**void**)** Returns the binary log of the page size.

unsigned **NSRoundDownToMultipleOfPageSize(**unsigned *byteCount***)**
Returns the multiple of the page size that is closest to, but not greater than, *byteCount*.

unsigned **NSRoundUpToMultipleOfPageSize(**unsigned *byteCount***)**
Returns the multiple of the page size that is closest to, but not less than, *byteCount*.

Get the amount of real memory:

unsigned **NSRealMemoryAvailable(**void**)** Returns the number of bytes available in the RAM hardware.

Allocate or free virtual memory:

void ***NSAllocateMemoryPages(**unsigned *byteCount***)**
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.

void **NSDeallocateMemoryPages(**void ***pointer*,
unsigned *byteCount***)** — Deallocates memory that was allocated with **NSAllocateMemoryPages()**.

void **NSCopyMemoryPages(**const void ***source*,
void ***destination*,
unsigned *byteCount***)** — Copies (or copies-on-write*) byteCount* bytes from *source* to *destination*.

Get a zone:

NSZone ***NSCreateZone(**unsigned *startSize*,
unsigned *granularity*,
BOOL *canFree***)** — Create 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.

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

NSZone ***NSZoneFromPointer(**void ***pointer***)** — Returns the zone for the *pointer* block of memory, 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.

Allocate or free memory in a zone:

void ***NSZoneMalloc(**NSZone ***zone*,
unsigned *size***)** — Allocates *size* bytes in *zone,* and returns a pointer to the allocated memory.

void ***NSZoneCalloc(**NSZone ***zone*,
unsigned *numElems*,
unsigned *numBytes***)** — Allocates enough memory from *zone* for *numElems* elements, each with a size of *numBytes* bytes, and returns a pointer to the allocated memory. The memory is initialized with zeros.

void ***NSZoneRealloc(**NSZone ***zone*,
void ***pointer*,
unsigned *size***)** — Changes the size of the block of memory pointed to by *pointer* 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. The *pointer* may be NULL.

void **NSRecycleZone(**NSZone ***zone***)** — Frees *zone* after adding any of its pointers still in use to the default zone.

(This strategy prevents retained objects from being inadvertently destroyed.)

void **NSZoneFree(**NSZone *_zone_, void *_pointer_**)** — 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.

Name a zone:

void **NSSetZoneName(**NSZone *_zone_, NSString *_name_**)** — Sets the specified zone's name to *name*, which can aid in debugging.

NSString ***NSZoneName(**NSZone *_zone_**)** — Returns the name of *zone*.

Object Allocation Functions

Allocate or free an object:

NSObject ***NSAllocateObject(**Class *aClass*, unsigned *extraBytes*, NSZone *_zone_**)** — Allocates and returns a pointer to an instance of *aClass*, 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.

NSObject ***NSCopyObject(**NSObject *_anObject_, unsigned *extraBytes*, NSZone *_zone_**)** — Creates and returns a new object that's an exact copy of *anObject*. The second and third arguments have the same meaning as in **NSAllocateObject()**.

void **NSDeallocateObject(**NSObject *_anObject_**)** — Deallocates *anObject*, which must have been allocated using **NSAllocateObject()**.

Decide whether to retain an object:

BOOL **NSShouldRetainWithZone(**NSObject *_anObject_, NSZone *_requestedZone_**)** — 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.

Modify the number of references to an object:

BOOL **NSDecrementExtraRefCountWasZero(**id *anObject***)**
Returns YES if the externally maintained ªextra reference countº for *anObject* is zero; otherwise, this function decrements the count and returns NO.

void **NSIncrementExtraRefCount(**id *anObject***)** Increments the externally maintained ªextra reference countº for *anObject*. The first reference (typically done in the **+alloc** method) isn't maintained externally, so there's no need to call this function for that first reference.

Error-Handling Functions

Change the top-level error handler:

NSUncaughtExceptionHandler ***NSGetUncaughtExceptionHandler(**void**)**
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.

void **NSSetUncaughtExceptionHandler(**NSUncaughtExceptionHandler **handler***)**
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.

Macros to handle an exception:

NS_DURING Marks the beginning of an exception-handling domain (a portion of code delimited by NS_DURING and NS_HANDLER). When an error is raised anywhere within the exception-handling domain, program execution jumps to the first line of code in the exception handler. It's illegal to exit

the exception-handling domain by any other means than NS_VALRETURN, NS_VOIDRETURN, or falling out the bottom.

NS_ENDHANDLER — Marks the ending of an exception handler (a portion of code delimited by NS_HANDLER and NS_ENDHANDLER).

NS_HANDLER — Marks the ending of an exception-handling domain and the beginning of the corresponding exception handler. Within the scope of the handler, a local variable called **exception** stores the raised exception. Code delimited by NS_HANDLER and NS_ENDHANDLER is never executed except when an error is raised in the preceding exception-handling domain.

value **NS_VALRETURN(***value***)** — Causes the method (or function) in which this macro occurs to return *value* immediately. This macro can only be placed within an exception-handling domain.

NS_VOIDRETURN — Causes the method (or function) in which this macro occurs to return immediately, with no return value. This macro can only be placed within an exception-handling domain.

Call the assertion handler from the body of an Objective-C method:

NSAssert(BOOL *condition,*
NSString **description***)* — Calls the NSAssertionHandler for the current thread if *condition* is false. The *description* should explain the error, formatted as for the standard C function **printf()**; it need not include the object's class and method name, since they're passed automatically to the handler.

NSAssert1(BOOL *condition,*
NSString **description,*
*arg***)** — Like **NSAssert()**, but the format string *description* includes a conversion specification (such as **%s** or **%d**) for the argument *arg*, in the style of **printf()**. You can pass an object in *arg* by specifying **%@**, which gets replaced by the string that the object's **description** method returns.

NSAssert2(BOOL *condition,*
NSString **description*,
arg1,
*arg2***)** — Like **NSAssert1()**, but with two arguments.

NSAssert3(BOOL *condition,*
NSString **description*,
arg1, — Like **NSAssert1()**, but with three arguments.

arg2,
arg3)

NSAssert4(BOOL *condition,*
NSString **description*,
arg1,
arg2,
arg3,
arg4)
Like **NSAssert1()**, but with four arguments.

NSAssert5(BOOL *condition,*
NSString **description*,
arg1,
arg2,
arg3,
arg4,
arg5)
Like **NSAssert1()**, but with five arguments.

Call the assertion handler from the body of a C function:

NSCAssert(BOOL *condition,*
NSString **description*)
Calls the NSAssertionHandler for the current thread if *condition* is false. The *description* should explain the error, formatted as for the standard C function **printf()**; it need not include the function name, which is passed automatically to the handler.

NSCAssert1(BOOL *condition,*
NSString **description,*
arg)
Like **NSCAssert()**, but the format string *description* includes a conversion specification (such as **%s** or **%d**) for the argument *arg*, in the style of **printf()**.

NSCAssert2(BOOL *condition,*
NSString **description*,
arg1,
arg2)
Like **NSCAssert1()**, but with two arguments.

NSCAssert3(BOOL *condition,*
NSString **description*,
arg1,
arg2,
arg3)
Like **NSCAssert1()**, but with three arguments.

NSCAssert4(BOOL *condition,*
NSString **description,*
arg1,
arg2,
arg3,
*arg4***)**
Like **NSCAssert1()**, but with four arguments.

NSCAssert5(BOOL *condition,*
NSString **description*,
arg1,
arg2,
arg3,
arg4,
*arg5***)**
Like **NSCAssert1()**, but with five arguments.

Validate a parameter:

NSParameterAssert(BOOL *condition***)**
Like **NSAssert()**, but the description passed to the assertion handler is ªInvalid parameter not satisfying: º followed by the text of *condition* (which can be any boolean expression).

NSCParameterAssert(BOOL *condition***)**
Like **NSParameterAssert()**, but to be called from the body of a C function.

Geometric Functions

Get a rectangle's coordinates:

float **NSMaxX(**NSRect *aRect***)**
Returns the largest x-coordinate value within *aRect.*

float **NSMaxY(**NSRect *aRect***)**
Returns the largest y-coordinate value within *aRect*.

float **NSMidX(**NSRect *aRect***)**
Returns the x-coordinate of the rectangle's center point.

float **NSMidY(**NSRect *aRect***)**
Returns the y-coordinate of the rectangle's center point.

float **NSMinX(**NSRect *aRect***)**
Returns the smallest x-coordinate value within *aRect*.

float **NSMinY(**NSRect *aRect***)**
Returns the smallest y-coordinate value within*aRect .*

float **NSWidth(**NSRect *aRect***)** — Returns the width of *aRect*.

float **NSHeight(**NSRect *aRect***)** — Returns the height of *aRect*.

Modify a copy of a rectangle:

NSRect **NSInsetRect(**NSRect *aRect*,
float *dX*,
float *dY***)** — 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*.

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

void **NSDivideRect(**NSRect *inRect*,
NSRect **slice*,
NSRect **remainder*,
float *amount*,
NSRectEdge *edge***)** — 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*Ð either NSMinXEdge, NSMinYEdge, NSMaxXEdge, or NSMaxYEdge). The size of *slice* is determined by *amount*, which measures the distance from *edge*.

NSRect **NSIntegralRect(**NSRect *aRect***)** — 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.

Compute a third rectangle from two rectangles:

NSRect **NSUnionRect(**NSRect *aRect*,
NSRect *bRect***)** — 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.

NSRect **NSIntersectionRect(**NSRect *aRect*,
NSRect *bRect***)** — 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.)

Test geometric relationships:

BOOL **NSEqualRects(**NSRect *aRect*, NSRect *bRect***)**	Returns YES if the two rectangles *aRect* and *bRect* are identical, and NO otherwise.
BOOL **NSEqualSizes(**NSSize *aSize*, NSSize *bSize***)**	Returns YES if the two sizes *aSize* and *bSize* are identical, and NO otherwise.
BOOL **NSEqualPoints(**NSPoint *aPoint*, NSPoint *bPoint***)**	Returns YES if the two points *aPoint* and *bPoint* are identical, and NO otherwise.
BOOL **NSIsEmptyRect(**NSRect *aRect***)**	Returns YES if the rectangle encloses no area at allÐthat is, if its width or height is zero or negative.
BOOL **NSMouseInRect(**NSPoint *aPoint*, NSRect *aRect*, BOOL *flipped***)**	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.
BOOL **NSPointInRect(**NSPoint *aPoint*, NSRect *aRect***)**	Performs the same test as **NSMouseInRect()**, but assumes a flipped coordinate system.
BOOL **NSContainsRect(**NSRect *aRect*, NSRect *bRect***)**	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.

Get a string representation:

NSString ***NSStringFromPoint(**NSPoint *aPoint***)**	Returns a string of the form ª{x=*a*; y=*b*}º, where *a* and *b* are the x- and y-coordinates of *aPoint*.
NSString ***NSStringFromRect(**NSRect *aRect***)**	Returns a string of the form ª{x=*a*; y=*b*; width=*c*; height=*d*}º, where *a, b, c,* and *d* are the x- and y-coordinates and the width and height, respectively, of *aRect*.
NSString ***NSStringFromSize(**NSSize *aSize***)**	Returns a string of the form ª{width=*a*; height=*b*}º, where *a* and *b* are the width and height of *aSize*.

Range Functions

Query a range:

BOOL **NSEqualRanges(**NSRange *range1*, NSRange *range2***)**	Returns YES if *range1* and *range2* have the same locations and lengths.
unsigned **NSMaxRange(**NSRange *range***)**	Returns *range*.location + *range*.lengthÐin other words, the number one greater than the maximum value within the range.
BOOL **NSLocationInRange(**unsigned *location*, NSRange *range***)**	Returns YES if *location* is in *range* (that is, if *location* is greater than or equal to *range*.location and *location* is less than **NSMaxRange(***range***)**).

Compute a range from two other ranges:

NSRange **NSUnionRange(**NSRange *range1*, NSRange *range2***)**	Returns a range whose maximum value is the greater of *range1*'s and *range2*'s maximum values, and whose location is the lesser of the two range's locations.
NSRange **NSIntersectionRange(**NSRange *range1*, NSRange *range2***)**	Returns a range whose maximum value is the lesser of *range1*'s and *range2*'s maximum values, and whose location is the greater of the two range's locations. However, if the two ranges don't intersect, the returned range has a location and length of zero.

Get a string representation:

NSString ***NSStringFromRange(**NSRange *range***)**	Returns a string of the form: ª{location = *a*; length = *b*}º, where *a* and *b* are non-negative integers.

Hash Table Functions

Create a table:

NSHashTable ***NSCreateHashTable(**NSHashTableCallBacks *callBacks*,
unsigned *capacity***)**

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 ªTypes and Constantsº), 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.

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

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

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

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.

Free a table:

void **NSFreeHashTable(**NSHashTable **table***)*

Releases each element of the specified hash table and frees the table itself.

void **NSResetHashTable(**NSHashTable **table***)*

Releases each element but doesn't deallocate the table. This is useful for preserving the table's capacity.

Compare two tables:

BOOL **NSCompareHashTables(**NSHashTable **table1*,
NSHashTable **table2***)** Returns YES if the two hash tables are equalÐthat is, if each element of *table1* is in *table2*, and the two tables are the same size.

Get the number of items:

unsigned **NSCountHashTable(**NSHashTable **table***)** Returns the number of elements in *table*.

Retrieve items:

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

NSArray ***NSAllHashTableObjects(**NSHashTable **table***)**
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.

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

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

Add or remove an item:

void **NSHashInsert(**NSHashTable **table*,
const void **pointer***)** 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.

void **NSHashInsertKnownAbsent(**NSHashTable **table*,
const void **pointer***)** Inserts *pointer*, which must not be NULL, into *table*. Unike **NSHashInsert()**,

this function raises an exception if *table* already includes an element that matches *pointer.*

void ***NSHashInsertIfAbsent(**NSHashTable **table*,
const void **pointer***)*

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

void **NSHashRemove(**NSHashTable **table*,
const void **pointer***)*

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

Get a string representation:

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

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.

Map Table Functions

Create a table:

NSMapTable ***NSCreateMapTable(**NSMapTableKeyCallBacks *keyCallBacks*,
NSMapTableValueCallBacks *valueCallBacks*,
unsigned *capacity***)**

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 ªTypes and Constantsº) that are very similar to the call-back structure used by **NSCreateHashTable()**; in fact, they have the same defaults as documented for that function.

NSMapTable ***NSCreateMapTableWithZone(**NSMapTableKeyCallBacks *keyCallBacks*,
NSMapTableValueCallBacks *valueCallBacks*,
unsigned *capacity*,
NSZone **zone***)*

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

NSMapTable ***NSCopyMapTableWithZone(**NSMapTable **table*,
NSZone **zone***)*

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.

Free a table:

void **NSFreeMapTable(**NSMapTable **table***)*

Releases each key and value of the specified map table and frees the table itself**.**

void **NSResetMapTable(**NSMapTable **table***)*

Releases each key and value but doesn't deallocate the table. This is useful for preserving the table's capacity.

Compare two tables:

BOOL **NSCompareMapTables(**NSMapTable **table1*,
NSMapTable **table2***)*

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.

Get the number of items:

unsigned **NSCountMapTable(**NSMapTable **table***)*

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

Retrieve items:

BOOL **NSMapMember(**NSMapTable **table*,
const void **key*,
void ***originalKey*,
void ***value***)**

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.*

void ***NSMapGet(**NSMapTable ***table**,
const void ***key**)
Returns the value that *table* maps to *key*, or NULL if the table doesn't contain *key*.

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

BOOL **NSNextMapEnumeratorPair(**NSMapEnumerator ***enumerator**,
void ****key**,
void ****value**)
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.

NSArray ***NSAllMapTableKeys(**NSMapTable ***table**)
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.

NSArray ***NSAllMapTableValues(**NSMapTable ***table**)
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.

Add or remove an item:

void **NSMapInsert(**NSMapTable ***table**,
const void ***key**,
const void ***value**)
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. An exception is raised if *key* is equal to the **notAKeyMarker** field of the table's NSMapTableKeyCallBacks structure.

void ***NSMapInsertIfAbsent(**NSMapTable ***table**,
const void ***key**,
const void ***value**)
If *key* matches a key already in *table*, this function returns the pre-existing key; otherwise, it adds *key* and *value* to the table and returns NULL.

void **NSMapInsertKnownAbsent(**NSMapTable ***table**,
const void ***key**,
const void ***value**)
Inserts *key* (which must not be **notAKeyMarker**) and *value* into *table*. Unike **NSMapInsert()**, this function raises an exception

if *table* already includes a key that matches *key.*

void **NSMapRemove(**NSMapTable ***table*, const void ***key***)**
If *key* matches a key already in *table*, this function releases the pre-existing key and its corresponding value.

NSString ****NSStringFromMapTable(**NSMapTable ***table***)**
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*a* = *b*;\nº, 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.

Miscellaneous Functions

Get information about a user:

NSString ****NSUserName(**void**)**
NSString ****NSHomeDirectory(**void**)**
NSString ****NSHomeDirectoryForUser(**NSString * *userName***)**

Log an error message:

void **NSLog(**NSString ***format*, ...**)**
Writes to **stderr** an error message of the form: ª*time processName processID formatº.* The *format* argument to **NSLog()** is a format string in the style of the standard C function **printf()**, followed by an arbitrary number of arguments that match conversion specifications (such as **%s** or **%d**) in the format string. (You can pass an object in the list of arguments by specifying **%@** in the format stringÐthis conversion specification gets replaced by the string that the object's **description** method returns.)

void **NSLogv(**NSString ***format*, va_list *args***)**
Like **NSLog()**, but the arguments to the format string are passed in a single va_list, in the manner of **vprintf()**.

Get localized versions of strings

NSString ***NSLocalizedString(**NSString **key*, NSString **comment**)
Returns a localized version of the string designated by *key*. The default string table (**Localizable.strings**) in the main bundle is searched for *key*. *comment* is ignored, but can provide information for translators**.**

NSString ***NSLocalizedStringFromTable(**NSString **key*, NSString **tableName*, NSString **comment**)
Like **NSLocalizedString()**, but searches the specified table.

NSString ***NSLocalizedStringFromTableInBundle(**NSString **key*, NSString **tableName*, NSBundle **aBundle*, NSString **comment**)
Like **NSLocalizedStringFromTable**, but uses the specified bundle instead of the application's main bundle.

Convert to and from a string:

Class **NSClassFromString(**NSString **aClassName**)
Returns the class object named by *aClassName*, or **nil** if none by this name is currently loaded*.*

SEL **NSSelectorFromString(**NSString **aSelectorName**)
Returns the selector named by *aSelectorName*, or zero if none by this name exists*.*

NSString ***NSStringFromClass(**Class *aClass***)** Returns an NSString containing the name of *aClass**.***

NSString ***NSStringFromSelector(**SEL *aSelector***)** Returns an NSString containing the name of *aSelector**.***

Compose a message to be sent later to an object

NSInvocation ***NS_INVOCATION**(Class *aClass*, *instanceMessage*)
Returns an NSInvocation object which you can later ask to dispatch *instanceMessage* to an instance of *aClass*. (You later use NSInvocation's **setTarget:** method to make a specific instance of *aClass* the receiver of the message, after which you use **invoke** to cause the message to be sent and **getReturnValue:** to retrieve the result.) Because this is a

macro, *message* can be any Objective C message understood by an instance of *aClass*, even a message with multiple arguments.

NSInvocation ***NS_MESSAGE**(id *anObject*, *instanceMessage*)

Like **NS_INVOCATION()**, but the first argument is an instance of a class, rather than a class. The target of the message will be *anObject,* so later you don't use **setTarget:**, only **invoke** and **getReturnValue:**.