Storage

Inherits From: Object

Declared In: objc/Storage.h

Class Description

The Storage class implements a general storage allocator. Each Storage object manages an array containing data elements of an arbitrary type. All the elements must be of the same type. When an element is added to the Storage object, it's copied into the array. The array grows dynamically when necessary; its capacity doesn't need to be explicitly adjusted.

Because a Storage object holds elements of an arbitrary type, you don't have to define a special class for each type of data you want to store. When setting up a new instance of the class, you specify the size of the elements and a description of their type. The type description is needed for archiving the object and must agree with the specified element size. It's encoded in a string using the descriptor codes listed in the table below:

Type	Code	Type	Code
int	i	**char**	c
unsigned int	I	**unsigned char**	C
short	s	**char** *	*
unsigned short	S	NXAtom	%
long	l	**id**	@
unsigned long	L	Class	#
float	f	SEL	:
double	d	structure	{<*types*>}
ignored	!	array	[<*count*><*types*>]

For example, "[15d]" means that each stored element is an array of fifteen **doubles**, and "{csi*@}" means that each stored element is a structure containing a **char**, a **short**, an **int**, a character pointer, and an object.

Most of these codes are identical to ones that would be returned by the **@encode()** compiler directive. However, there are some differences:

- A structure description can contain only encoded type information between the braces. It can't include a full type name or structure name.
- The '%' descriptor specifies a unique string pointer. When the pointer is unarchived, the **NXUniqueString()** function is called to make sure that it's also unique within the new context.
- The '!' descriptor marks data that won't be archived. Each occurrence of '!' instructs the archiver to skip data the size of an **int**.
- A few **@encode()** descriptors—such as the ones for pointers, bitfields, and undefined types—should not be used. Use only the codes shown in the table above.

Instance Variables

void ***dataPtr**;
const char ***description**;
unsigned int **numElements**;
unsigned int **maxElements**;
unsigned int **elementSize**;

dataPtr	A pointer to the data stored by the object.
description	A string encoding the type of data stored.
numElements	The number of elements actually in the Storage array.
maxElements	The total number of elements that can fit within currently allocated memory.
elementSize	The size of each element in the array.

Method Types

Initializing a new Storage instance
- – init
- – initCount:elementSize:description:

Copying and freeing Storage objects
- – copyFromZone:
- – free

Getting, adding, and removing elements
- – addElement:
- – insertElement:at:
- – removeElementAt:
- – removeLastElement
- – replaceElementAt:with:
- – empty
- – elementAt:

Comparing Storage objects
- – isEqual:

Managing the storage capacity and type
- – count
- – description
- – setAvailableCapacity:
- – setNumSlots:

Archiving
- – read:
- – write:

Instance Methods

addElement:

– **addElement:**(void *)*anElement*

Adds *anElement* at the end of the Storage array and returns **self**. The size of the array is increased if necessary.

See also: – **insertElement:at:**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Returns a new Storage object containing the same data as the receiver. The data and the object are both copied, and memory for both is taken from *zone*. However, the description string is not copied; the two objects share the same string.

See also: – **copy** (Object)

count

– (unsigned int)**count**

Returns the number of elements currently in the Storage array.

See also: – **setNumSlots:**

description

– (const char *)**description**

Returns the string encoding the data type of elements in the Storage array.

See also: – **initCount:elementSize:description:**

elementAt:

– (void *)**elementAt:**(unsigned int)*index*

Returns a pointer to the element at *index* in the Storage array. If no element is stored at *index* (*index* is beyond the end of the array), a NULL pointer is returned.

Before using the pointer that's returned, you must convert it into the appropriate type by a cast. The pointer can be used either to read the element at *index* or to alter it.

See also: – **replaceElementAt:with:**, – **insertElement:at:**

empty

– **empty**

Empties the Storage array of all its elements and returns **self**. The current capacity of the array remains unchanged; nothing is deallocated.

See also: – **free**

free

– **free**

Frees the Storage object and all the elements it contains. Pointers stored in the object will be freed, but the data they point to won't be (unless the data is also stored in the object). You might want to free the data before freeing the Storage object. The description string isn't freed.

See also: – **empty**

init

– **init**

Initializes the Storage object so that it's ready to store object **id**s. The initial capacity of the array isn't set. In general, it's better to store object **id**s in a List object. Returns **self**.

See also: – **initCount:elementSize:description:**, – **initCount:** (List)

initCount:elementSize:description:

– **initCount:**(unsigned int)*count*
elementSize:(unsigned int)*sizeInBytes*
description:(const char *)*string*

Initializes the Storage object so that it has *count* elements. Each element is of size *sizeInBytes* and of the type described by *string*. Memory for all the elements is set to 0. Returns **self**.

If *string* is NULL, the object won't be archivable. Once set, the description string should never be modified.

This method is the designated initializer for the class. It's used to initialize Storage objects immediately after they have been allocated; it should never be used to reinitialize a Storage object that's already been placed in use.

insertElement:at:

– **insertElement:**(void *)*anElement* **at:**(unsigned int)*index*

Puts *anElement* in the Storage array at *index*. All elements between *index* and the last element are shifted to make room. The size of the array is increased if necessary. Returns **self**.

See also: – **addElement:**, – **setNumSlots:**

isEqual:

– (BOOL)**isEqual:***anObject*

Compares the receiver with *anObject*, and returns YES if they're the same and NO if they're not. Two Storage objects are considered to be the same if they have the same number of elements and the elements at each position in the array match.

read:

– **read:**(NXTypedStream *)*stream*

Reads the Storage object and the data it stores from the typed stream *stream*. Where an archived string is represented by a '%' descriptor, the **NXUniqueString()** function is called to make sure that the string is unique within the new context.

See also: – **write:**

removeElementAt:

– **removeElementAt:**(unsigned int)*index*

Removes the element located at *index* from the Storage array and returns **self**. All elements between *index* and the last element are shifted to close the gap.

See also: – **removeLastElement**

removeLastElement

– **removeLastElement**

Removes the last element from the Storage array and returns **self**.

See also: – **removeElementAt:**

replaceElementAt:with:

– **replaceElementAt:**(unsigned int)*index* **with:**(void *)*anElement*

Replaces the data at *index* with the data pointed to by *anElement*. However, if no element is stored at *index* (*index* is beyond the end of the array), nothing is replaced. Returns **self**.

See also: – **elementAt:**, – **insertElement:at:**

setAvailableCapacity:

– **setAvailableCapacity:**(unsigned int)*numSlots*

Sets the storage capacity of the array to at least *numSlots* elements and returns **self**. If the array already contains more than *numSlots* elements, its capacity is left unchanged and **nil** is returned.

See also: – **setNumSlots:**, – **count**

setNumSlots:

– **setNumSlots:**(unsigned int)*numSlots*

Sets the number of elements in the Storage array to *numSlots* and returns **self**. If *numSlots* is greater than the current number of elements in the array (the value returned by **count**), the new slots will be filled with zeros. If *numSlots* is less than the current number of elements in the array, access to all elements with indices equal to or greater than *numSlots* will be lost.

If necessary, this method increases the capacity of the storage array so there's room for at least *numSlots* elements.

See also: – **setAvailableCapacity:**, – **count**

write:

– **write:**(NXTypedStream *)*stream*

Writes the Storage object and its data to the typed stream *stream*.

See also: – **read:**