Programming Language—Common Lisp

15. Arrays

15.1 Array Concepts

15.1.1 Array Elements

An *array* contains a set of *objects* called *elements* that can be referenced individually according to a rectilinear coordinate system.

15.1.1.1 Array Indices

An *array element* is referred to by a (possibly empty) series of indices. The length of the series must equal the *rank* of the *array*. Each index must be a non-negative *fixnum* less than the corresponding *array dimension*. *Array* indexing is zero-origin.

15.1.1.2 Array Dimensions

An axis of an *array* is called a ***dimension***.

Each *dimension* is a non-negative *fixnum*; if any dimension of an *array* is zero, the *array* has no elements. It is permissible for a *dimension* to be zero, in which case the *array* has no elements, and any attempt to *access* an *element* is an error. However, other properties of the *array*, such as the *dimensions* themselves, may be used.

15.1.1.2.1 Implementation Limits on Individual Array Dimensions

An *implementation* may impose a limit on *dimensions* of an *array*, but there is a minimum requirement on that limit. See the *variable* **array-dimension-limit**.

15.1.1.3 Array Rank

An *array* can have any number of *dimensions* (including zero). The number of *dimensions* is called the ***rank***.

If the rank of an *array* is zero then the *array* is said to have no *dimensions*, and the product of the dimensions (see **array-total-size**) is then 1; a zero-rank *array* therefore has a single element.

15.1.1.3.1 Vectors

An *array* of *rank* one (*i.e.*, a one-dimensional *array*) is called a ***vector***.

15.1.1.3.1.1 Fill Pointers

A ***fill pointer*** is a non-negative *integer* no larger than the total number of *elements* in a *vector*. Not all *vectors* have *fill pointers*. See the *functions* **make-array** and **adjust-array**.

An *element* of a *vector* is said to be ***active*** if it has an index that is greater than or equal to zero, but less than the *fill pointer* (if any). For an *array* that has no *fill pointer*, all *elements* are considered *active*.

Only *vectors* may have *fill pointers*; multidimensional *arrays* may not. A multidimensional *array* that is displaced to a *vector* that has a *fill pointer* can be created.

15.1.1.3.2 Multidimensional Arrays

15.1.1.3.2.1 Storage Layout for Multidimensional Arrays

Multidimensional *arrays* store their components in row-major order; that is, internally a multidimensional *array* is stored as a one-dimensional *array*, with the multidimensional index sets ordered lexicographically, last index varying fastest.

15.1.1.3.2.2 Implementation Limits on Array Rank

An *implementation* may impose a limit on the *rank* of an *array*, but there is a minimum requirement on that limit. See the *variable* **array-rank-limit**.

15.1.2 Specialized Arrays

An *array* can be a *general array*, meaning each *element* may be any *object*, or it may be a *specialized array*, meaning that each *element* must be of a restricted *type*.

The phrasing "an *array specialized* to *type* ⟨⟨*type*⟩⟩" is sometimes used to emphasize the *element type* of an *array*. This phrasing is tolerated even when the ⟨⟨*type*⟩⟩ is **t**, even though an *array specialized* to *type t* is a *general array*, not a *specialized array*.

Figure 15–1 lists some *defined names* that are applicable to *array* creation, *access*, and information operations.

adjust-array	**array-in-bounds-p**	**svref**
adjustable-array-p	**array-rank**	**upgraded-array-element-type**
aref	**array-rank-limit**	**upgraded-complex-part-type**
array-dimension	**array-row-major-index**	**vector**
array-dimension-limit	**array-total-size**	**vector-pop**
array-dimensions	**array-total-size-limit**	**vector-push**
array-element-type	**fill-pointer**	**vector-push-extend**
array-has-fill-pointer-p	**make-array**	

Figure 15–1. General Purpose Array-Related Defined Names

15.1.2.1 Array Upgrading

The ***upgraded array element type*** of a *type* T_1 is a *type* T_2 that is a *supertype* of T_1 and that is used instead of T_1 whenever T_1 is used as an *array element type* for object creation or type discrimination.

During creation of an *array*, the *element type* that was requested is called the ***expressed array element type***. The *upgraded array element type* of the *expressed array element type* becomes the ***actual array element type*** of the *array* that is created.

Type upgrading implies a movement upwards in the type hierarchy lattice. A *type* is always a *subtype* of its *upgraded array element type*. Also, if a *type* T_x is a *subtype* of another *type* T_y, then the *upgraded array element type* of T_x must be a *subtype* of the *upgraded array element type* of T_y. Two *disjoint types* can be *upgraded* to the same *type*.

The *upgraded array element type* T_2 of a *type* T_1 is a function only of T_1 itself; that is, it is independent of any other property of the *array* for which T_2 will be used, such as *rank*, *adjustability*, *fill pointers*, or displacement. The *function* **upgraded-array-element-type** can be used by *conforming programs* to predict how the *implementation* will *upgrade* a given *type*.

15.1.2.2 Required Kinds of Specialized Arrays

Vectors whose *elements* are restricted to *type* **character** or a *subtype* of **character** are called ***strings***. *Strings* are of *type* **string**. Figure 15–2 lists some *defined names* related to *strings*.

Strings are *specialized arrays* and might logically have been included in this chapter. However, for purposes of readability most information about *strings* does not appear in this chapter; see instead Chapter 16 (Strings).

char	**string-equal**	**string-upcase**
make-string	**string-greaterp**	**string/=**
nstring-capitalize	**string-left-trim**	**string<**
nstring-downcase	**string-lessp**	**string<=**
nstring-upcase	**string-not-equal**	**string=**
schar	**string-not-greaterp**	**string>**
string	**string-not-lessp**	**string>=**
string-capitalize	**string-right-trim**	
string-downcase	**string-trim**	

Figure 15–2. Operators that Manipulate Strings

Vectors whose *elements* are restricted to *type* **bit** are called ***bit vectors***. *Bit vectors* are of *type* **bit-vector**. Figure 15–3 lists some *defined names* for operations on *bit arrays*.

bit	**bit-ior**	**bit-orc2**
bit-and	**bit-nand**	**bit-xor**
bit-andc1	**bit-nor**	**sbit**
bit-andc2	**bit-not**	
bit-eqv	**bit-orc1**	

Figure 15–3. Operators that Manipulate Bit Arrays

array *System Class*

Class Precedence List:

array, **t**

Description:

An *array* contains *objects* arranged according to a Cartesian coordinate system. An *array* provides mappings from a set of *fixnums* $\{i_0, i_1, \ldots, i_{r-1}\}$ to corresponding *elements* of the *array*, where $0 \leq i_j < d_j$, r is the rank of the array, and d_j is the size of *dimension* j of the array.

When an *array* is created, the program requesting its creation may declare that all *elements* are of a particular *type*, called the *expressed array element type*. The implementation is permitted to *upgrade* this type in order to produce the *actual array element type*, which is the *element type* for the *array* is actually *specialized*. See the *function* **upgraded-array-element-type**.

Compound Type Specifier Kind:

Specializing.

Compound Type Specifier Syntax:

(array [{*element-type* | *} [*dimension-spec*]])

dimension-spec::=*rank* | * | ({*dimension* | *}*)

Compound Type Specifier Arguments:

dimension—a *valid array dimension*.

element-type—a *type specifier*.

rank—a non-negative *fixnum*.

Compound Type Specifier Description:

This denotes the set of *arrays* whose *element type*, *rank*, and *dimensions* match any given *element-type*, *rank*, and *dimensions*. Specifically:

If *element-type* is the *symbol* *, *arrays* are not excluded on the basis of their *element type*. Otherwise, only those *arrays* are included whose *actual array element type* is the result of *upgrading* *element-type*; see Section 15.1.2.1 (Array Upgrading).

If the *dimension-spec* is a *rank*, the set includes only those *arrays* having that *rank*. If the *dimension-spec* is a *list* of *dimensions*, the set includes only those *arrays* having a *rank* given by the *length* of the *dimensions*, and having the indicated *dimensions*; in this case, * matches any value for the corresponding *dimension*. If the *dimension-spec* is the *symbol* *, the set is not restricted on the basis of *rank* or *dimension*.

See Also:

print-array, **aref**, **make-array**, **vector**, Section 2.4.8.12 (Sharpsign A), Section 22.1.3.11 (Printing Other Arrays)

Notes:

Note that the type `(array t)` is a proper *subtype* of the type `(array *)`. The reason is that the type `(array t)` is the set of *arrays* that can hold any *object* (the *elements* are of *type* `t`, which includes all *objects*). On the other hand, the type `(array *)` is the set of all *arrays* whatsoever, including for example *arrays* that can hold only *characters*. The type `(array character)` is not a *subtype* of the type `(array t)`; the two sets are *disjoint* because the type `(array character)` is not the set of all *arrays* that can hold *characters*, but rather the set of *arrays* that are specialized to hold precisely *characters* and no other *objects*.

simple-array *Type*

Supertypes:

simple-array, **array**, **t**

Description:

The *type* of an *array* that is not displaced to another *array*, has no *fill pointer*, and is not *expressly adjustable* is a *subtype* of *type* **simple-array**. The concept of a *simple array* exists to allow the implementation to use a specialized representation and to allow the user to declare that certain values will always be *simple arrays*.

The *types* **simple-vector**, **simple-string**, and **simple-bit-vector** are *disjoint subtypes* of *type* **simple-array**, for they respectively mean `(simple-array t (*))`, the union of all `(simple-array` *c* `(*))` for any *c* being a *subtype* of *type* **character**, and `(simple-array bit (*))`.

Compound Type Specifier Kind:

Specializing.

Compound Type Specifier Syntax:

`(simple-array` [{*element-type* | *} [*dimension-spec*]]`)`

dimension-spec::=*rank* | * | ({*dimension* | *}*)

Compound Type Specifier Arguments:

dimension—a *valid array dimension*.

element-type—a *type specifier*.

rank—a non-negative *fixnum*.

Compound Type Specifier Description:

This *compound type specifier* is treated exactly as the corresponding *compound type specifier* for *type* **array** would be treated, except that the set is further constrained to include only *simple arrays*.

Notes:

It is *implementation-dependent* whether *displaced arrays*, *vectors* with *fill pointers*, or arrays that are *actually adjustable* are *simple arrays*.

`(simple-array *)` refers to all *simple arrays* regardless of element type, `(simple-array` *type-specifier*`)` refers only to those *simple arrays* that can result from giving *type-specifier* as the `:element-type` argument to **make-array**.

vector *System Class*

Class Precedence List:

vector, **array**, **sequence**, **t**

Description:

Any one-dimensional *array* is a *vector*.

The *type* **vector** is a *subtype* of *type* **array**; for all *types* `x`, `(vector x)` is the same as `(array x (*))`.

The *type* `(vector t)`, the *type* **string**, and the *type* **bit-vector** are *disjoint subtypes* of *type* **vector**.

Compound Type Specifier Kind:

Specializing.

Compound Type Specifier Syntax:

`(vector` [{*element-type* | *} [{*size* | *}]]`)`

Compound Type Specifier Arguments:

size—a non-negative *fixnum*.

element-type—a *type specifier*.

Compound Type Specifier Description:

This denotes the set of specialized *vectors* whose *element type* and *dimension* match the specified values. Specifically:

If *element-type* is the *symbol* *, *vectors* are not excluded on the basis of their *element type*. Otherwise, only those *vectors* are included whose *actual array element type* is the result of *upgrading* *element-type*; see Section 15.1.2.1 (Array Upgrading).

If a *size* is specified, the set includes only those *vectors* whose only *dimension* is *size*. If the *symbol* * is specified instead of a *size*, the set is not restricted on the basis of *dimension*.

See Also:

Section 15.1.2.2 (Required Kinds of Specialized Arrays), Section 2.4.8.3 (Sharpsign Left-Parenthesis), Section 22.1.3.10 (Printing Other Vectors), Section 2.4.8.12 (Sharpsign A)

Notes:

The *type* `(vector e s)` is equivalent to the *type* `(array e (s))`.

The type `(vector bit)` has the name **bit-vector**.

The union of all *types* `(vector` C`)`, where C is any *subtype* of **character**, has the name **string**.

`(vector *)` refers to all *vectors* regardless of element type, `(vector type-specifier)` refers only to those *vectors* that can result from giving *type-specifier* as the `:element-type` argument to **make-array**.

simple-vector *Type*

Supertypes:

simple-vector, **vector**, **simple-array**, **array**, **sequence**, **t**

Description:

The *type* of a *vector* that is not displaced to another *array*, has no *fill pointer*, is not *expressly adjustable* and is able to hold elements of any *type* is a *subtype* of *type* **simple-vector**.

The *type* **simple-vector** is a *subtype* of *type* **vector**, and is a *subtype* of *type* `(vector t)`.

Compound Type Specifier Kind:

Specializing.

Compound Type Specifier Syntax:

`(simple-vector [size])`

Compound Type Specifier Arguments:

size—a non-negative *fixnum*, or the *symbol* *. The default is the *symbol* *.

Compound Type Specifier Description:

This is the same as `(simple-array t (`*size*`))`.

bit-vector *System Class*

Class Precedence List:

bit-vector, **vector**, **array**, **sequence**, **t**

Description:

A *bit vector* is a *vector* the *element type* of which is *bit*.

The *type* **bit-vector** is a *subtype* of *type* **vector**, for **bit-vector** means `(vector bit)`.

Compound Type Specifier Kind:

Abbreviating.

Compound Type Specifier Syntax:

`(bit-vector [`*size*`])`

Compound Type Specifier Arguments:

size—a non-negative *fixnum*, or the *symbol* `*`.

Compound Type Specifier Description:

This denotes the same *type* as the *type* `(array bit (`*size*`))`; that is, the set of *bit vectors* of size *size*.

See Also:

Section 2.4.8.4 (Sharpsign Asterisk), Section 22.1.3.9 (Printing Bit Vectors), Section 15.1.2.2 (Required Kinds of Specialized Arrays)

simple-bit-vector *Type*

Supertypes:

simple-bit-vector, **bit-vector**, **vector**, **simple-array**, **array**, **sequence**, **t**

Description:

The *type* of a *bit vector* that is not displaced to another *array*, has no *fill pointer*, and is not *expressly adjustable* is a *subtype* of *type* **simple-bit-vector**.

Compound Type Specifier Kind:

Abbreviating.

Compound Type Specifier Syntax:

`(simple-bit-vector [`*size*`])`

Compound Type Specifier Arguments:

size—a non-negative *fixnum*, or the *symbol* `*`. The default is the *symbol* `*`.

Compound Type Specifier Description:

This denotes the same type as the *type* `(simple-array bit (`*size*`))`; that is, the set of *simple bit vectors* of size *size*.

make-array *Function*

Syntax:

make-array *dimensions* `&key` *element-type*
initial-element
initial-contents
adjustable
fill-pointer
displaced-to
displaced-index-offset

→ *new-array*

Arguments and Values:

dimensions—a *designator* for a *list* of *valid array dimensions*.

element-type—a *type specifier*. The default is `t`.

initial-element—an *object*.

make-array

initial-contents—an *object*.

adjustable—a *boolean*. The default is **nil**.

fill-pointer—a *valid fill pointer* for the *array* to be created, or **t** or **nil**. The default is **nil**.

displaced-to—an *array* or **nil**. The default is **nil**. This option must not be supplied if either *initial-element* or *initial-contents* is supplied.

displaced-index-offset—a *valid array row-major index* for *displaced-to*. The default is `0`. This option must not be supplied unless a *non-nil* *displaced-to* is supplied.

new-array—an *array*.

Description:

Creates and returns an *array* constructed of the most *specialized type* that can accommodate elements of *type* given by *element-type*. If *dimensions* is **nil** then a zero-dimensional *array* is created.

Dimensions represents the dimensionality of the new *array*.

element-type indicates the *type* of the elements intended to be stored in the *new-array*. The *new-array* can actually store any *objects* of the *type* which results from *upgrading* *element-type*; see Section 15.1.2.1 (Array Upgrading).

If *initial-element* is supplied, it is used to initialize each element of *array*. If *initial-element* is supplied, it must be of the *type* given by *element-type*. *initial-element* cannot be supplied if either the `:initial-contents` option is supplied or *displaced-to* is *non-nil*. If *initial-element* is not supplied, the initial values of the *array elements* are *implementation-dependent* unless either *initial-contents* is supplied or *displaced-to* is *non-nil*.

initial-contents is used to initialize the contents of *array*. For example:

```
(make-array '(4 2 3) :initial-contents
            '(((a b c) (1 2 3))
             ((d e f) (3 1 2))
             ((g h i) (2 3 1))
             ((j k l) (0 0 0))))
```

initial-contents is composed of a nested structure of *sequences*. The numbers of levels in the structure must equal the rank of *array*. Each leaf of the nested structure must be of the *type* given by *element-type*. If *array* is zero-dimensional, then *initial-contents* specifies the single *element*. Otherwise, *initial-contents* must be a *sequence* whose length is equal to the first dimension; each element must be a nested structure for an *array* whose dimensions are the remaining dimensions, and so on. *Initial-contents* cannot be supplied if either *initial-element* is supplied or *displaced-to* is *non-nil*. If *initial-contents* is not supplied, the initial values of the *array elements* are *implementation-dependent* unless either *initial-element* is supplied or *displaced-to* is *non-nil*.

If *adjustable* is *non-nil*, the array is *expressly adjustable* (and so *actually adjustable*); otherwise,

make-array

the array is not *expressly adjustable* (and it is *implementation-dependent* whether the array is *actually adjustable*).

If *fill-pointer* is *non-nil*, the *array* must be one-dimensional; that is, the *array* must be a *vector*. If *fill-pointer* is t, the length of the *vector* is used to initialize the *fill pointer*. If *fill-pointer* is an *integer*, it becomes the initial *fill pointer* for the *vector*.

If *displaced-to* is *non-nil*, **make-array** will create a *displaced array* and *displaced-to* is the *target* of that *displaced array*. In that case, the consequences are undefined if the *actual array element type* of *displaced-to* is not *type equivalent* to the *actual array element type* of the *array* being created. If *displaced-to* is **nil**, the *array* is not a *displaced array*.

The *displaced-index-offset* is made to be the index offset of the *array*. When an array A is given as the `:displaced-to` *argument* to **make-array** when creating array B, then array B is said to be displaced to array A. The total number of elements in an *array*, called the total size of the *array*, is calculated as the product of all the dimensions. It is required that the total size of A be no smaller than the sum of the total size of B plus the offset `n` supplied by the *displaced-index-offset*. The effect of displacing is that array B does not have any elements of its own, but instead maps *accesses* to itself into *accesses* to array A. The mapping treats both *arrays* as if they were one-dimensional by taking the elements in row-major order, and then maps an *access* to element `k` of array B to an *access* to element `k+n` of array A.

If **make-array** is called with *adjustable*, *fill-pointer*, and *displaced-to* each **nil**, then the result is a *simple array*. If **make-array** is called with one or more of *adjustable*, *fill-pointer*, or *displaced-to* being *true*, whether the resulting *array* is a *simple array* is *implementation-dependent*.

When an array A is given as the `:displaced-to` *argument* to **make-array** when creating array B, then array B is said to be displaced to array A. The total number of elements in an *array*, called the total size of the *array*, is calculated as the product of all the dimensions. The consequences are unspecified if the total size of A is smaller than the sum of the total size of B plus the offset `n` supplied by the *displaced-index-offset*. The effect of displacing is that array B does not have any elements of its own, but instead maps *accesses* to itself into *accesses* to array A. The mapping treats both *arrays* as if they were one-dimensional by taking the elements in row-major order, and then maps an *access* to element `k` of array B to an *access* to *element* `k+n` of array A.

Examples:

```
(make-array 5) ;; Creates a one-dimensional array of five elements.
(make-array '(3 4) :element-type '(mod 16)) ;; Creates a
              ;;two-dimensional array, 3 by 4, with four-bit elements.
(make-array 5 :element-type 'single-float) ;; Creates an array of single-floats.

(make-array nil :initial-element nil) → #0ANIL
(make-array 4 :initial-element nil) → #(NIL NIL NIL NIL)
(make-array '(2 4)
             :element-type '(unsigned-byte 2)
```

make-array

```
             :initial-contents '((0 1 2 3) (3 2 1 0)))
→ #2A((0 1 2 3) (3 2 1 0))
 (make-array 6
             :element-type 'character
             :initial-element #\a
             :fill-pointer 3) → "aaa"
```

The following is an example of making a *displaced array*.

```
 (setq a (make-array '(4 3)))
→ #<ARRAY 4x3 simple 32546632>
 (dotimes (i 4)
   (dotimes (j 3)
     (setf (aref a i j) (list i 'x j '= (* i j)))))
→ NIL
 (setq b (make-array 8 :displaced-to a
                       :displaced-index-offset 2))
→ #<ARRAY 8 indirect 32550757>
 (dotimes (i 8)
   (print (list i (aref b i))))
▷ (0 (0 X 2 = 0))
▷ (1 (1 X 0 = 0))
▷ (2 (1 X 1 = 1))
▷ (3 (1 X 2 = 2))
▷ (4 (2 X 0 = 0))
▷ (5 (2 X 1 = 2))
▷ (6 (2 X 2 = 4))
▷ (7 (3 X 0 = 0))
→ NIL
```

The last example depends on the fact that *arrays* are, in effect, stored in row-major order.

```
 (setq a1 (make-array 50))
→ #<ARRAY 50 simple 32562043>
 (setq b1 (make-array 20 :displaced-to a1 :displaced-index-offset 10))
→ #<ARRAY 20 indirect 32563346>
 (length b1) → 20

 (setq a2 (make-array 50 :fill-pointer 10))
→ #<ARRAY 50 fill-pointer 10 46100216>
 (setq b2 (make-array 20 :displaced-to a2 :displaced-index-offset 10))
→ #<ARRAY 20 indirect 46104010>
 (length a2) → 10
 (length b2) → 20
```

```
 (setq a3 (make-array 50 :fill-pointer 10))
→ #<ARRAY 50 fill-pointer 10 46105663>
 (setq b3 (make-array 20 :displaced-to a3 :displaced-index-offset 10
                         :fill-pointer 5))
→ #<ARRAY 20 indirect, fill-pointer 5 46107432>
 (length a3) → 10
 (length b3) → 5
```

See Also:

adjustable-array-p, **aref**, **arrayp**, **array-element-type**, **array-rank-limit**, **array-dimension-limit**, **fill-pointer**, **upgraded-array-element-type**

Notes:

There is no specified way to create an *array* for which **adjustable-array-p** definitely returns *false*. There is no specified way to create an *array* that is not a *simple array*.

adjust-array *Function*

Syntax:

adjust-array *array new-dimensions* &key *element-type*
initial-element
initial-contents
fill-pointer
displaced-to
displaced-index-offset

→ *adjusted-array*

Arguments and Values:

array—an *array*.

new-dimensions—a *valid array dimension* or a *list* of *valid array dimensions*.

element-type—a *type specifier*.

initial-element—an *object*. *Initial-element* must not be supplied if either *initial-contents* or *displaced-to* is supplied.

initial-contents—an *object*. If *array* has rank greater than zero, then *initial-contents* is composed of nested *sequences*, the depth of which must equal the rank of *array*. Otherwise, *array* is zero-dimensional and *initial-contents* supplies the single element. *initial-contents* must not be supplied if either *initial-element* or *displaced-to* is given.

fill-pointer—a *valid fill pointer* for the *array* to be created, or **t**, or **nil**. The default is **nil**.

adjust-array

displaced-to—an *array* or **nil**. *initial-elements* and *initial-contents* must not be supplied if *displaced-to* is supplied.

displaced-index-offset—an *object* of *type* `(fixnum 0 n)` where n is `(array-total-size` *displaced-to*`)`. *displaced-index-offset* may be supplied only if *displaced-to* is supplied.

adjusted-array—an *array*.

Description:

adjust-array changes the dimensions or elements of *array*. The result is an *array* of the same *type* and rank as *array*, that is either the modified *array*, or a newly created *array* to which *array* can be displaced, and that has the given *new-dimensions*.

New-dimensions specify the size of each *dimension* of *array*.

Element-type specifies the *type* of the *elements* of the resulting *array*. If *element-type* is supplied, the consequences are unspecified if the *upgraded array element type* of *element-type* is not the same as the *actual array element type* of *array*.

If *initial-contents* is supplied, it is treated as for **make-array**. In this case none of the original contents of *array* appears in the resulting *array*.

If *fill-pointer* is an *integer*, it becomes the *fill pointer* for the resulting *array*. If *fill-pointer* is the symbol **t**, it indicates that the size of the resulting *array* should be used as the *fill pointer*. If *fill-pointer* is **nil**, it indicates that the *fill pointer* should be left as it is.

If *displaced-to* *non-nil*, a *displaced array* is created. The resulting *array* shares its contents with the *array* given by *displaced-to*. The resulting *array* cannot contain more elements than the *array* it is displaced to. If *displaced-to* is not supplied or **nil**, the resulting *array* is not a *displaced array*. If array A is created displaced to array B and subsequently array B is given to **adjust-array**, array A will still be displaced to array B. Although *array* might be a *displaced array*, the resulting *array* is not a *displaced array* unless *displaced-to* is supplied and not **nil**. The interaction between **adjust-array** and displaced *arrays* is as follows given three *arrays*, `A`, `B`, and `C`:

`A` is not displaced before or after the call

```
(adjust-array A ...)
```

The dimensions of `A` are altered, and the contents rearranged as appropriate. Additional elements of `A` are taken from *initial-element*. The use of *initial-contents* causes all old contents to be discarded.

`A` is not displaced before, but is displaced to `C` after the call

```
(adjust-array A ... :displaced-to C)
```

None of the original contents of `A` appears in `A` afterwards; `A` now contains the contents of `C`, without any rearrangement of `C`.

adjust-array

A is displaced to B before the call, and is displaced to C after the call

```
(adjust-array A ... :displaced-to B)
(adjust-array A ... :displaced-to C)
```

B and C might be the same. The contents of B do not appear in A afterward unless such contents also happen to be in C If *displaced-index-offset* is not supplied in the **adjust-array** call, it defaults to zero; the old offset into B is not retained.

A is displaced to B before the call, but not displaced afterward.

```
(adjust-array A ... :displaced-to B)
(adjust-array A ... :displaced-to nil)
```

A gets a new "data region," and contents of B are copied into it as appropriate to maintain the existing old contents; additional elements of A are taken from *initial-element* if supplied. However, the use of *initial-contents* causes all old contents to be discarded.

If *displaced-index-offset* is supplied, it specifies the offset of the resulting *array* from the beginning of the *array* that it is displaced to. If *displaced-index-offset* is not supplied, the offset is 0. The size of the resulting *array* plus the offset value cannot exceed the size of the *array* that it is displaced to.

If only *new-dimensions* and an *initial-element* argument are supplied, those elements of *array* that are still in bounds appear in the resulting *array*. The elements of the resulting *array* that are not in the bounds of *array* are initialized to *initial-element*; if *initial-element* is not provided, then the initial contents of any resulting *elements* is *implementation-dependent*.

If *initial-contents* or *displaced-to* is supplied, then none of the original contents of *array* appears in the new *array*.

The consequences are unspecified if *array* is adjusted to a size smaller than its *fill pointer* without supplying the *fill-pointer* argument so that its *fill-pointer* is properly adjusted in the process.

If A is displaced to B, the consequences are unspecified if B is adjusted in such a way that it no longer has enough elements to satisfy A.

If **adjust-array** is applied to an *array* that is *actually adjustable*, the *array* returned is *identical* to *array*. If the *array* returned by **adjust-array** is *distinct* from *array*, then the argument *array* is unchanged.

Note that if an *array* A is displaced to another *array* B, and B is displaced to another *array* C, and B is altered by **adjust-array**, A must now refer to the adjust contents of B. This means that an implementation cannot collapse the chain to make A refer to C directly and forget that the chain of reference passes through B. However, caching techniques are permitted as long as they preserve the semantics specified here.

Examples:

```
 (adjustable-array-p
  (setq ada (adjust-array
              (make-array '(2 3)
                          :adjustable t
                          :initial-contents '((a b c) (1 2 3)))
              '(4 6)))) → T
 (array-dimensions ada) → (4 6)
 (aref ada 1 1) → 2
 (setq beta (make-array '(2 3) :adjustable t))
→ #2A((NIL NIL NIL) (NIL NIL NIL))
 (adjust-array beta '(4 6) :displaced-to ada)
→ #2A((A B C NIL NIL NIL)
      (1 2 3 NIL NIL NIL)
      (NIL NIL NIL NIL NIL NIL)
      (NIL NIL NIL NIL NIL NIL))
 (array-dimensions beta) → (4 6)
 (aref beta 1 1) → 2
```

Suppose that the 4-by-4 array in `m` looks like this:

```
#2A(( alpha     beta      gamma     delta )
    ( epsilon   zeta      eta       theta )
    ( iota      kappa     lambda    mu    )
    ( nu        xi        omicron   pi    ))
```

Then the result of

```
 (adjust-array m '(3 5) :initial-element 'baz)
```

is a 3-by-5 array with contents

```
#2A(( alpha     beta      gamma     delta     baz )
    ( epsilon   zeta      eta       theta     baz )
    ( iota      kappa     lambda    mu        baz ))
```

Exceptional Situations:

An error of *type* **error** is signaled if *fill-pointer* is supplied and *non-nil* but *array* has no *fill pointer*.

See Also:

adjustable-array-p, **make-array**, **array-dimension-limit**, **array-total-size-limit**, **array**

adjustable-array-p *Function*

Syntax:

adjustable-array-p *array* → *boolean*

Arguments and Values:

array—an *array*.

boolean—a *boolean*.

Description:

Returns true if and only if **adjust-array** could return a *value* which is *identical* to *array* when given that *array* as its first *argument*.

Examples:

```
(adjustable-array-p
  (make-array 5
              :element-type 'character
              :adjustable t
              :fill-pointer 3)) → true
(adjustable-array-p (make-array 4)) → implementation-dependent
```

Exceptional Situations:

Should signal an error of *type* **type-error** if its argument is not an *array*.

See Also:

adjust-array, **make-array**

aref *Accessor*

Syntax:

aref *array* `&rest` *subscripts* → *element*

(**setf** (**aref** *array* `&rest` *subscripts*) *new-element*)

Arguments and Values:

array—an *array*.

subscripts—a *list* of *valid array indices* for the *array*.

element, *new-element*—an *object*.

Description:

Accesses the *array element* specified by the *subscripts*. If no *subscripts* are supplied and *array* is zero rank, **aref** *accesses* the sole element of *array*.

aref ignores *fill pointers*. It is permissible to use **aref** to *access* any *array element*, whether *active* or not.

Examples:

If the variable `foo` names a 3-by-5 array, then the first index could be 0, 1, or 2, and then second index could be 0, 1, 2, 3, or 4. The array elements can be referred to by using the *function* **aref**; for example, `(aref foo 2 1)` refers to element (2, 1) of the array.

```
(aref (setq alpha (make-array 4)) 3) → implementation-dependent
(setf (aref alpha 3) 'sirens) → SIRENS
(aref alpha 3) → SIRENS
(aref (setq beta (make-array '(2 4)
                 :element-type '(unsigned-byte 2)
                 :initial-contents '((0 1 2 3) (3 2 1 0))))
      1 2) → 1
(setq gamma '(0 2))
(apply #'aref beta gamma) → 2
(setf (apply #'aref beta gamma) 3) → 3
(apply #'aref beta gamma) → 3
(aref beta 0 2) → 3
```

See Also:

bit, **char**, **elt**, **row-major-aref**, **svref**, Section 3.2.1 (Terminology)

array-dimension *Function*

Syntax:

array-dimension *array axis-number* → *dimension*

Arguments and Values:

array—an *array*.

axis-number—an *integer* greater than or equal to zero and less than the *rank* of the *array*.

dimension—a non-negative *integer*.

Description:

array-dimension returns the *axis-number* $dimension_1$ of *array*. (Any *fill pointer* is ignored.)

Examples:

```
(array-dimension (make-array 4) 0) → 4
(array-dimension (make-array '(2 3)) 1) → 3
```

Affected By:

None.

See Also:

array-dimensions, **length**

Notes:

```
(array-dimension array n) ≡ (nth n (array-dimensions array))
```

array-dimensions *Function*

Syntax:

array-dimensions *array* → *dimensions*

Arguments and Values:

array—an *array*.

dimensions—a *list* of *integers*.

Description:

Returns a *list* of the *dimensions* of *array*. (If *array* is a *vector* with a *fill pointer*, that *fill pointer* is ignored.)

Examples:

```
(array-dimensions (make-array 4)) → (4)
(array-dimensions (make-array '(2 3))) → (2 3)
(array-dimensions (make-array 4 :fill-pointer 2)) → (4)
```

Exceptional Situations:

Should signal an error of *type* **type-error** if its argument is not an *array*.

See Also:

array-dimension

Notes:

array-element-type *Function*

Syntax:

array-element-type *array* → *typespec*

Arguments and Values:

array—an *array*.

typespec—a *type specifier*.

Description:

Returns a *type specifier* which represents the *actual array element type* of the array, which is the set of *objects* that such an *array* can hold. (Because of *array upgrading*, this *type specifier* can in some cases denote a *supertype* of the *expressed array element type* of the *array*.)

Examples:

```
 (array-element-type (make-array 4)) → T
 (array-element-type (make-array 12 :element-type '(unsigned-byte 8)))
→ implementation-dependent
 (array-element-type (make-array 12 :element-type '(unsigned-byte 5)))
→ implementation-dependent
```

```
 (array-element-type (make-array 5 :element-type '(mod 5)))
```

could be `(mod 5)`, `(mod 8)`, `fixnum`, `t`, or any other type of which `(mod 5)` is a *subtype*.

Affected By:

The *implementation*.

Exceptional Situations:

Should signal an error of *type* **type-error** if its argument is not an *array*.

See Also:

array, **make-array**, **subtypep**, **upgraded-array-element-type**

array-has-fill-pointer-p *Function*

Syntax:

array-has-fill-pointer-p *array* → *boolean*

Arguments and Values:

array—an *array*.

boolean—a *boolean*.

Description:

Returns *true* if *array* has a *fill pointer*; otherwise returns *false*.

Examples:

```
(array-has-fill-pointer-p (make-array 4)) → implementation-dependent
(array-has-fill-pointer-p (make-array '(2 3))) → false
(array-has-fill-pointer-p
  (make-array 8
              :fill-pointer 2
              :initial-element 'filler)) → true
```

Exceptional Situations:

Should signal an error of *type* **type-error** if its argument is not an *array*.

See Also:

make-array, **fill-pointer**

Notes:

Since *arrays* of *rank* other than one cannot have a *fill pointer*, **array-has-fill-pointer-p** always returns **nil** when its argument is such an array.

array-displacement *Function*

Syntax:

array-displacement *array* → *displaced-to, displaced-index-offset*

Arguments and Values:

array—an *array*.

displaced-to—an *array* or **nil**.

displaced-index-offset—a non-negative *fixnum*.

Description:

If the *array* is a *displaced array*, returns the *values* of the `:displaced-to` and `:displaced-index-offset` options for the *array* (see the *functions* **make-array** and **adjust-array**). If the *array* is not a *displaced array*, **nil** and `0` are returned.

If **array-displacement** is called on an *array* for which a *non-nil object* was provided as the `:displaced-to` *argument* to **make-array** or **adjust-array**, it must return that *object* as its first value. It is *implementation-dependent* whether **array-displacement** returns a *non-nil primary value* for any other *array*.

Examples:

```
 (setq a1 (make-array 5)) → #<ARRAY 5 simple 46115576>
 (setq a2 (make-array 4 :displaced-to a1
                        :displaced-index-offset 1))
→ #<ARRAY 4 indirect 46117134>
 (array-displacement a2)
→ #<ARRAY 5 simple 46115576>, 1
 (setq a3 (make-array 2 :displaced-to a2
                        :displaced-index-offset 2))
→ #<ARRAY 2 indirect 46122527>
 (array-displacement a3)
→ #<ARRAY 4 indirect 46117134>, 2
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *array* is not an *array*.

See Also:

make-array

Notes:

array-in-bounds-p *Function*

Syntax:

array-in-bounds-p *array* `&rest` *subscripts* → *boolean*

Arguments and Values:

array—an *array*.

subscripts—a list of *integers* of length equal to the *rank* of the *array*.

boolean—a *boolean*.

Description:

Returns *true* if the *subscripts* are all in bounds for *array*; otherwise returns *false*. (If *array* is a *vector* with a *fill pointer*, that *fill pointer* is ignored.)

Examples:

```
(setq a (make-array '(7 11) :element-type 'string-char))
(array-in-bounds-p a 0  0) → true
(array-in-bounds-p a 6 10) → true
(array-in-bounds-p a 0 -1) → false
(array-in-bounds-p a 0 11) → false
(array-in-bounds-p a 7  0) → false
```

See Also:

array-dimensions

Notes:

```
(array-in-bounds-p array subscripts)
≡ (and (not (some #'minusp (list subscripts)))
       (every #'< (list subscripts) (array-dimensions array)))
```

array-rank *Function*

Syntax:

array-rank *array* → *rank*

Arguments and Values:

array—an *array*.

rank—a non-negative *integer*.

Description:

Returns the number of *dimensions* of *array*.

Examples:

```
(array-rank (make-array '())) → 0
(array-rank (make-array 4)) → 1
(array-rank (make-array '(4))) → 1
(array-rank (make-array '(2 3))) → 2
```

Exceptional Situations:

Should signal an error of *type* **type-error** if its argument is not an *array*.

See Also:

array-rank-limit, **make-array**

array-row-major-index *Function*

Syntax:

array-row-major-index *array* &rest *subscripts* → *index*

Arguments and Values:

array—an *array*.

subscripts—a *list* of *valid array indices* for the *array*.

index—a *valid array row-major index* for the *array*.

Description:

Computes the position according to the row-major ordering of *array* for the element that is specified by *subscripts*, and returns the offset of the element in the computed position from the beginning of *array*.

For a one-dimensional *array*, the result of **array-row-major-index** equals *subscript*.

array-row-major-index ignores *fill pointers*.

Examples:

```
(setq a (make-array '(4 7) :element-type '(unsigned-byte 8)))
(array-row-major-index a 1 2) → 9
(array-row-major-index
   (make-array '(2 3 4)
               :element-type '(unsigned-byte 8)
               :displaced-to a
               :displaced-index-offset 4)
   0 2 1) → 9
```

Notes:

A possible definition of **array-row-major-index**, with no error-checking, is

```
(defun array-row-major-index (a &rest subscripts)
  (apply #'+ (maplist #'(lambda (x y)
                          (* (car x) (apply #'* (cdr y))))
```

```
subscripts
(array-dimensions a))))
```

array-total-size *Function*

Syntax:

array-total-size *array* → *size*

Arguments and Values:

array—an *array*.

size—a non-negative *integer*.

Description:

Returns the *array total size* of the *array*.

Examples:

```
(array-total-size (make-array 4)) → 4
(array-total-size (make-array 4 :fill-pointer 2)) → 4
(array-total-size (make-array 0)) → 0
(array-total-size (make-array '(4 2))) → 8
(array-total-size (make-array '(4 0))) → 0
(array-total-size (make-array '())) → 1
```

Exceptional Situations:

Should signal an error of *type* **type-error** if its argument is not an *array*.

See Also:

make-array, **array-dimensions**

Notes:

If the *array* is a *vector* with a *fill pointer*, the *fill pointer* is ignored when calculating the *array total size*.

Since the product of no arguments is one, the *array total size* of a zero-dimensional *array* is one.

```
(array-total-size x)
   ≡ (apply #'* (array-dimensions x))
   ≡ (reduce #'* (array-dimensions x))
```

arrayp *Function*

Syntax:

arrayp *object* → *boolean*

Arguments and Values:

object—an *object*.

boolean—a *boolean*.

Description:

Returns *true* if *object* is of *type* **array**; otherwise, returns *false*.

Examples:

```
(arrayp (make-array '(2 3 4) :adjustable t)) → true
(arrayp (make-array 6)) → true
(arrayp #*1011) → true
(arrayp "hi") → true
(arrayp 'hi) → false
(arrayp 12) → false
```

See Also:

typep

Notes:

```
(arrayp object) ≡ (typep object 'array)
```

fill-pointer *Accessor*

Syntax:

fill-pointer *vector* → *fill-pointer*

(**setf** (**fill-pointer** *vector*) *new-fill-pointer*)

Arguments and Values:

vector—a *vector* with a *fill pointer*.

fill-pointer, *new-fill-pointer*—a *valid fill pointer* for the *vector*.

Description:

Accesses the *fill pointer* of *vector*.

Examples:

```
(setq a (make-array 8 :fill-pointer 4)) → #(NIL NIL NIL NIL)
(fill-pointer a) → 4
(dotimes (i (length a)) (setf (aref a i) (* i i))) → NIL
a → #(0 1 4 9)
(setf (fill-pointer a) 3) → 3
(fill-pointer a) → 3
a → #(0 1 4)
(setf (fill-pointer a) 8) → 8
a → #(0 1 4 9 NIL NIL NIL NIL)
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *vector* is not a *vector* with a *fill pointer*.

See Also:

make-array, **length**

Notes:

There is no *operator* that will remove a *vector*'s *fill pointer*.

row-major-aref *Accessor*

Syntax:

row-major-aref *array index* → *element*

(**setf** (**row-major-aref** *array index*) *new-element*)

Arguments and Values:

array—an *array*.

index—a *valid array row-major index* for the *array*.

element, *new-element*—an *object*.

Description:

Considers *array* as a *vector* by viewing its *elements* in row-major order, and returns the *element* of that *vector* which is referred to by the given *index*.

row-major-aref is valid for use with **setf**.

See Also:

aref, **array-row-major-index**

Notes:

```
(row-major-aref array index) ≡
  (aref (make-array (array-total-size array)
                    :displaced-to array
                    :element-type (array-element-type array))
        index)

(aref array i1 i2 ...) ≡
    (row-major-aref array (array-row-major-index array i1 i2))
```

upgraded-array-element-type *Function*

Syntax:

upgraded-array-element-type *typespec* `&optional` *environment* → *upgraded-typespec*

Arguments and Values:

typespec—a *type specifier*.

environment—an *environment object*. The default is **nil**, denoting the *null lexical environment* and the current *global environment*.

upgraded-typespec—a *type specifier*.

Description:

Returns the *element type* of the most *specialized array* representation capable of holding items of the *type* denoted by *typespec*.

The *typespec* is a *subtype* of (and possibly *type equivalent* to) the *upgraded-typespec*.

If *typespec* is **bit**, the result is *type equivalent* to `bit`. If *typespec* is **base-char**, the result is *type equivalent* to `base-char`. If *typespec* is **character**, the result is *type equivalent* to `character`.

The purpose of **upgraded-array-element-type** is to reveal how an implementation does its *upgrading*.

The *environment* is used to expand any *derived type specifiers* that are mentioned in the *typespec*.

See Also:

array-element-type, **make-array**

Notes:

Except for storage allocation consequences and dealing correctly with the optional *environment argument*, **upgraded-array-element-type** could be defined as:

```
(defun upgraded-array-element-type (type &optional environment)
  (array-element-type (make-array 0 :element-type type)))
```

array-dimension-limit *Constant Variable*

Constant Value:

A positive *fixnum*, the exact magnitude of which is *implementation-dependent*, but which is not less than `1024`.

Description:

The upper exclusive bound on each individual *dimension* of an *array*.

See Also:

make-array

array-rank-limit *Constant Variable*

Constant Value:

A positive *fixnum*, the exact magnitude of which is *implementation-dependent*, but which is not less than `8`.

Description:

The upper exclusive bound on the *rank* of an *array*.

See Also:

make-array

array-total-size-limit *Constant Variable*

Constant Value:

A positive *fixnum*, the exact magnitude of which is *implementation-dependent*, but which is not less than `1024`.

Description:

The upper exclusive bound on the *array total size* of an *array*.

The actual limit on the *array total size* imposed by the *implementation* might vary according the *element type* of the *array*; in this case, the value of **array-total-size-limit** will be the smallest of these possible limits.

See Also:

make-array, **array-element-type**

simple-vector-p *Function*

Syntax:

simple-vector-p *object* → *boolean*

Arguments and Values:

object—an *object*.

boolean—a *boolean*.

Description:

Returns *true* if *object* is of *type* **simple-vector**; otherwise, returns *false*..

Examples:

```
(simple-vector-p (make-array 6)) → true
(simple-vector-p "aaaaaa") → false
(simple-vector-p (make-array 6 :fill-pointer t)) → false
```

See Also:

simple-vector

Notes:

```
(simple-vector-p object) ≡ (typep object 'simple-vector)
```

svref *Accessor*

Syntax:

svref *simple-vector index* → *element*

(**setf** (**svref** *simple-vector index*) *new-element*)

Arguments and Values:

simple-vector—a *simple vector*.

index—a *valid array index* for the *simple-vector*.

element, *new-element*—an *object* (whose *type* is a *subtype* of the *array element type* of the *simple-vector*).

Description:

Accesses the *element* of *simple-vector* specified by *index*.

Examples:

```
(simple-vector-p (setq v (vector 1 2 'sirens))) → true
(svref v 0) → 1
(svref v 2) → SIRENS
(setf (svref v 1) 'newcomer) → NEWCOMER
v → #(1 NEWCOMER SIRENS)
```

See Also:

aref, **sbit**, **schar**, **vector**, Section 3.2.1 (Terminology)

Notes:

svref is identical to **aref** except that it requires its first argument to be a *simple vector*.

```
(svref v i) ≡ (aref (the simple-vector v) i)
```

vector *Function*

Syntax:

vector &rest *objects* → *vector*

Arguments and Values:

object—an *object*.

vector—a *vector* of *type* (vector t *).

Description:

Creates a *fresh simple general vector* whose size corresponds to the number of *objects*.

The *vector* is initialized to contain the *objects*.

Examples:

```
(arrayp (setq v (vector 1 2 'sirens))) → true
(vectorp v) → true
(simple-vector-p v) → true
(length v) → 3
```

See Also:

make-array

Notes:

vector is analogous to **list**.

```
(vector a1 a2 ... an)
 ≡ (make-array (list n) :element-type t
                        :initial-contents
                          (list a1 a2 ... an))
```

vector-pop *Function*

Syntax:

vector-pop *vector* → *element*

Arguments and Values:

vector—a *vector* with a *fill pointer*.

element—an *object*.

Description:

Decreases the *fill pointer* of *vector* by one, and retrieves the *element* of *vector* that is designated by the new *fill pointer*.

Examples:

```
(vector-push (setq fable (list 'fable))
             (setq fa (make-array 8
                                  :fill-pointer 2
                                  :initial-element 'sisyphus))) → 2
(fill-pointer fa) → 3
(eq (vector-pop fa) fable) → true
(vector-pop fa) → SISYPHUS
(fill-pointer fa) → 1
```

Side Effects:

The *fill pointer* is decreased by one.

Affected By:

The value of the *fill pointer*.

Exceptional Situations:

An error of *type* **type-error** is signaled if *vector* does not have a *fill pointer*.

If the *fill pointer* is zero, **vector-pop** signals an error of *type* **error**.

See Also:

vector-push, **vector-push-extend**, **fill-pointer**

vector-push, vector-push-extend *Function*

Syntax:

vector-push *new-element vector* → *new-index-p*

vector-push-extend *new-element vector* `&optional` *extension* → *new-index*

Arguments and Values:

new-element—an *object*.

vector—a *vector* with a *fill pointer*.

extension—a positive *integer*. The default is *implementation-dependent*.

new-index-p—a *valid array index* for *vector*, or **nil**.

vector-push, vector-push-extend

new-index—a *valid array index* for *vector*.

Description:

vector-push and **vector-push-extend** store *new-element* in *vector*. **vector-push** attempts to store *new-element* in the element of *vector* designated by the *fill pointer*, and to increase the *fill pointer* by one. If the `(>= (fill-pointer vector) (array-dimension vector 0))`, neither *vector* nor its *fill pointer* are affected. Otherwise, the store and increment take place and **vector-push** returns the former value of the *fill pointer* which is one less than the one it leaves in *vector*.

vector-push-extend is just like **vector-push** except that if the *fill pointer* gets too large, *vector* is extended using **adjust-array** so that it can contain more elements. *Extension* is the minimum number of elements to be added to *vector* if it must be extended.

vector-push and **vector-push-extend** return the index of *new-element* in *vector*. If `(>= (fill-pointer vector) (array-dimension vector 0))`, **vector-push** returns **nil**.

Examples:

```
(vector-push (setq fable (list 'fable))
             (setq fa (make-array 8
                                  :fill-pointer 2
                                  :initial-element 'first-one))) → 2
(fill-pointer fa) → 3
(eq (aref fa 2) fable) → true
(vector-push-extend #\X
                   (setq aa
                         (make-array 5
                                     :element-type 'character
                                     :adjustable t
                                     :fill-pointer 3))) → 3
(fill-pointer aa) → 4
(vector-push-extend #\Y aa 4) → 4
(array-total-size aa) → at least 5
(vector-push-extend #\Z aa 4) → 5
(array-total-size aa) → 9 ;(or more)
```

Affected By:

The value of the *fill pointer*.

How *vector* was created.

Exceptional Situations:

An error of *type* **error** is signaled by **vector-push-extend** if it tries to extend *vector* and *vector* is not *actually adjustable*.

An error of *type* **error** is signaled if *vector* does not have a *fill pointer*.

See Also:

adjustable-array-p, **fill-pointer**, **vector-pop**

vectorp *Function*

Syntax:

vectorp *object* → *boolean*

Arguments and Values:

object—an *object*.

boolean—a *boolean*.

Description:

Returns *true* if *object* is of *type* **vector**; otherwise, returns *false*.

Examples:

```
(vectorp "aaaaaa") → true
(vectorp (make-array 6 :fill-pointer t)) → true
(vectorp (make-array '(2 3 4))) → false
(vectorp #*11) → true
(vectorp #b11) → false
```

Notes:

```
(vectorp object) ≡ (typep object 'vector)
```

bit, sbit *Accessor*

Syntax:

bit *bit-array* &rest *subscripts* → *bit*
sbit *bit-array* &rest *subscripts* → *bit*

(setf (**bit** *bit-array* &rest *subscripts*) *new-bit*)
(setf (**sbit** *bit-array* &rest *subscripts*) *new-bit*)

Arguments and Values:

bit-array—for **bit**, a *bit array*; for **sbit**, a *simple bit array*.

subscripts—a *list* of *valid array indices* for the *bit-array*.

bit—a *bit*.

Description:

bit and **sbit** *access* the *bit-array element* specified by *subscripts*.

These *functions* ignore the *fill pointer* when *accessing elements*.

Examples:

```
(bit (setq ba (make-array 8
                          :element-type 'bit
                          :initial-element 1))
      3) → 1
(setf (bit ba 3) 0) → 0
(bit ba 3) → 0
(sbit ba 5) → 1
(setf (sbit ba 5) 1) → 1
(sbit ba 5) → 1
```

See Also:

aref, Section 3.2.1 (Terminology)

Notes:

bit and **sbit** are like **aref** except that they require *arrays* to be a *bit array* and a *simple bit array*, respectively.

bit and **sbit**, unlike **char** and **schar**, allow the first argument to be an *array* of any *rank*.

bit-and, bit-andc1, bit-andc2, bit-eqv, bit-ior, bit-nand, bit-nor, bit-not, bit-orc1, bit-orc2, bit-xor

Function

Syntax:

bit-and *bit-array1 bit-array2* `&optional` *opt-arg* → *resulting-bit-array*
bit-andc1 *bit-array1 bit-array2* `&optional` *opt-arg* → *resulting-bit-array*
bit-andc2 *bit-array1 bit-array2* `&optional` *opt-arg* → *resulting-bit-array*
bit-eqv *bit-array1 bit-array2* `&optional` *opt-arg* → *resulting-bit-array*
bit-ior *bit-array1 bit-array2* `&optional` *opt-arg* → *resulting-bit-array*
bit-nand *bit-array1 bit-array2* `&optional` *opt-arg* → *resulting-bit-array*
bit-nor *bit-array1 bit-array2* `&optional` *opt-arg* → *resulting-bit-array*
bit-orc1 *bit-array1 bit-array2* `&optional` *opt-arg* → *resulting-bit-array*

bit-and, bit-andc1, bit-andc2, bit-eqv, bit-ior, ...

bit-orc2 *bit-array1 bit-array2* &optional *opt-arg* → *resulting-bit-array*
bit-xor *bit-array1 bit-array2* &optional *opt-arg* → *resulting-bit-array*

bit-not *bit-array* &optional *opt-arg* → *resulting-bit-array*

Arguments and Values:

bit-array, *bit-array1*, *bit-array2*—a *bit array*.

Opt-arg—a *bit array*, or t, or **nil**. The default is **nil**.

Bit-array, *bit-array1*, *bit-array2*, and *opt-arg* (if an *array*) must all be of the same *rank* and *dimensions*.

resulting-bit-array—a *bit array*.

Description:

These functions perform bit-wise logical operations on *bit-array1* and *bit-array2* and return an *array* of matching *rank* and *dimensions*, such that any given bit of the result is produced by operating on corresponding bits from each of the arguments.

In the case of **bit-not**, an *array* of *rank* and *dimensions* matching *bit-array* is returned that contains a copy of *bit-array* with all the bits inverted.

If *opt-arg* is of type (array bit) the contents of the result are destructively placed into *opt-arg*. If *opt-arg* is the symbol t, *bit-array* or *bit-array1* is replaced with the result; if *opt-arg* is **nil** or omitted, a new *array* is created to contain the result.

Figure 15–4 indicates the logical operation performed by each of the *functions*.

Function	Operation
bit-and	and
bit-eqv	equivalence (exclusive nor)
bit-not	complement
bit-ior	inclusive or
bit-xor	exclusive or
bit-nand	complement of *bit-array1* and *bit-array2*
bit-nor	complement of *bit-array1* or *bit-array2*
bit-andc1	and complement of *bit-array1* with *bit-array2*
bit-andc2	and *bit-array1* with complement of *bit-array2*
bit-orc1	or complement of *bit-array1* with *bit-array2*
bit-orc2	or *bit-array1* with complement of *bit-array2*

Figure 15–4. Bit-wise Logical Operations on Bit Arrays

Examples:

```
(bit-and (setq ba #*11101010) #*01101011) → #*01101010
(bit-and #*1100 #*1010) → #*1000
(bit-andc1 #*1100 #*1010) → #*0010
(setq rba (bit-andc2 ba #*00110011 t)) → #*11001000
(eq rba ba) → true
(bit-not (setq ba #*11101010)) → #*00010101
(setq rba (bit-not ba
                   (setq tba (make-array 8
                                         :element-type 'bit))))
→ #*00010101
(equal rba tba) → true
(bit-xor #*1100 #*1010) → #*0110
```

See Also:

lognot, **logand**

bit-vector-p *Function*

Syntax:

bit-vector-p *object* → *boolean*

Arguments and Values:

object—an *object*.

boolean—a *boolean*.

Description:

Returns *true* if *object* is of *type* **bit-vector**; otherwise, returns *false*.

Examples:

```
(bit-vector-p (make-array 6
                          :element-type 'bit
                          :fill-pointer t)) → true
(bit-vector-p #*) → true
(bit-vector-p (make-array 6)) → false
```

See Also:

typep

Notes:

```
(bit-vector-p object) ≡ (typep object 'bit-vector)
```

simple-bit-vector-p *Function*

Syntax:

simple-bit-vector-p *object* → *boolean*

Arguments and Values:

object—an *object*.

boolean—a *boolean*.

Description:

Returns *true* if *object* is of *type* **simple-bit-vector**; otherwise, returns *false*.

Examples:

```
(simple-bit-vector-p (make-array 6)) → false
(simple-bit-vector-p #*) → true
```

See Also:

simple-vector-p

Notes:

```
(simple-bit-vector-p object) ≡ (typep object 'simple-bit-vector)
```

Table of Contents