Mawk Arrays
rcs-version 1.3 2014/08/01

Table of Contents

1. Introduction. This is the source and documenation for the `mawk` implementation of awk arrays. Arrays in awk are associations of strings to awk scalar values. The mawk implementation stores the associations in hash tables. The hash table scheme was influenced by and is similar to the design presented in Griswold and Townsend, *The Design and Implementation of Dynamic Hashing Sets and Tables in Icon*, **Software Practice and Experience**, 23, 351-367, 1993.

2. Data Structures.

2.1. Array Structure. The type `ARRAY` is a pointer to a `struct array`. The `size` field is the number of elements in the table. The meaning of the other fields depends on the `type` field.

⟨ array typedefs and `#defines` 1a ⟩≡
```
typedef struct array {
   PTR ptr ;  /* What this points to depends on the type */
   size_t size ; /* number of elts in the table */
   size_t limit ; /* Meaning depends on type */
   unsigned hmask ; /* bitwise and with hash value to get table index */
   short type ;  /* values in AY_NULL .. AY_SPLIT */
} *ARRAY ;
```
See also 1b, 3a and 13b.

This code is used in 21d.

There are three types of arrays and these are distinguished by the `type` field in the structure. The types are:

`AY_NULL` The array is empty and the `size` field is always zero. The other fields have no meaning.

`AY_SPLIT` The array was created by the `AWK` built-in `split`. The return value from `split` is stored in the `size` field. The `ptr` field points at a vector of `CELLs`. The number of `CELLs` is the `limit` field. It is always true that $size \le limit$. The address of `A[i]` is `(CELL*)A->ptr+i-1` for $1 \le i \le size$. The `hmask` field has no meaning.

Hash Table The array is a hash table. If the `AY_STR` bit in the `type` field is set, then the table is keyed on strings. If the `AY_INT` bit in the `type` field is set, then the table is keyed on integers. Both bits can be set, and then the two keys are consistent, i.e., look up of `A[-14]` and `A["-14"]` will return identical `CELL` pointers although the look up methods will be different. In this case, the `size` field is the number of hash nodes in the table. When insertion of a new element would cause `size` to exceed `limit`, the table grows by doubling the number of hash chains. The invariant, $(hmask + 1)max\_ave\_list\_length = limit$, is always true. *Max_ave_list_length* is a tunable constant.

⟨ array typedefs and `#defines` 1b ⟩+≡
```
#define AY_NULL         0
#define AY_INT          1
#define AY_STR          2
#define AY_SPLIT        4
```
See also 1a, 3a and 13b.

This code is used in 21d.

2.2. Hash Tables. The hash tables are linked lists of nodes, called `ANODEs`. The number of lists is `hmask+1` which is always a power of two. The `ptr` field points at a vector of list heads. Since there are potentially two types of lists, integer lists and strings lists, each list head is a structure, `DUAL_LINK`.

⟨ local constants, defs and prototypes 1c ⟩≡
```
struct anode ;
typedef struct {struct anode *slink, *ilink ;} DUAL_LINK ;
```
See also 2a, 5b, 7b, 8a, 9a, 14a, 14c, 15b and 16b.

This code is used in 22a.

The string lists are chains connected by `slinks` and the integer lists are chains connected by `ilinks`. We sometimes refer to these lists as slists and ilists, respectively. The elements on the lists are `ANODEs`. The fields of an `ANODE` are:

`slink` The link field for slists.

`ilink` The link field for ilists.

`sval` If non-null, then `sval` is a pointer to a string key. For a given table, if the `AY_STR` bit is set then every `ANODE` has a non-null `sval` field and conversely, if `AY_STR` is not set, then every `sval` field is null.

`hval` The hash value of `sval`. This field has no meaning if `sval` is null.

`ival` The integer key. The field has no meaning if set to the constant, `NOT_AN_IVALUE`. If the `AY_STR` bit is off, then every `ANODE` will have a valid `ival` field. If the `AY_STR` bit is on, then the `ival` field may or may not be valid.

`cell` The data field in the hash table.

So the value of $A[expr]$ is stored in the `cell` field, and if *expr* is an integer, then *expr* is stored in `ival`, else it is stored in `sval`.

⟨ local constants, defs and prototypes 2a ⟩ +≡

```
typedef struct anode {
   struct anode *slink ;
   struct anode  *ilink ;
   STRING *sval ;
   unsigned hval ;
   Int     ival ;
   CELL    cell ;
} ANODE ;
```

See also 1c, 5b, 7b, 8a, 9a, 14a, 14c, 15b and 16b.

This code is used in 22a.

3. Array Operations. The functions that operate on arrays are,

`CELL* array_find(ARRAY A, CELL *cp, int create_flag)` returns a pointer to $A[expr]$ where `cp` is a pointer to the `CELL` holding *expr*. If the `create_flag` is on and *expr* is not an element of `A`, then the element is created with value *null*.

`void array_delete(ARRAY A, CELL *cp)` removes an element $A[expr]$ from the array A. `cp` points at the `CELL` holding *expr*.

`void array_load(ARRAY A, size_t cnt)` builds a split array. The values $A[1..cnt]$ are copied from the array $split\_buff[0..cnt-1]$.

`void array_clear(ARRAY A)` removes all elements of A. The type of A is then `AY_NULL`.

`STRING** array_loop_vector(ARRAY A, size_t *sizep)` returns a pointer to a linear vector that holds all the strings that are indices of A. The size of the the vector is returned indirectly in `*sizep`. If `A->size==0`, a *null* pointer is returned.

`CELL* array_cat(CELL *sp, int cnt)` concatenates the elements of $sp[1-cnt..0]$, with each element separated by `SUBSEP`, to compute an array index. For example, on a reference to $A[i,j]$, `array_cat` computes $i \circ SUBSEP \circ j$ where $\circ$ denotes concatenation.

⟨ interface prototypes 2b ⟩ ≡

```
CELL* array_find(ARRAY, CELL*, int);
void  array_delete(ARRAY, CELL*);
void  array_load(ARRAY, size_t);
void  array_clear(ARRAY);
STRING** array_loop_vector(ARRAY, size_t*);
CELL* array_cat(CELL*, int);
```

This code is used in 21d.

3.1. Array Find. Any reference to $A[expr]$ creates a call to `array_find(A,cp,CREATE)` where `cp` points at the cell holding *expr*. The test, *expr* in A, creates a call to `array_find(A,cp,NO_CREATE)`.

⟨ array typedefs and `#defines` 3a ⟩ +≡
```
#define NO_CREATE  0
#define CREATE     1
```

See also 1a, 1b and 13b.

This code is used in 21d.

`Array_find` is hash-table lookup that breaks into two cases:

(1) If `*cp` is numeric and integer valued, then lookup by integer value using `find_by_ival`. If `*cp` is numeric, but not integer valued, then convert to string with `sprintf(CONVFMT,...)` and go to case 2.

(2) If `*cp` is string valued, then lookup by string value using `find_by_sval`.

⟨ interface functions 3b ⟩ ≡
```
CELL* array_find(
   ARRAY A,
   CELL *cp,
   int create_flag)
{
   ANODE *ap ;
   int redid ;
   if (A->size == 0 && !create_flag)
      /* eliminating this trivial case early avoids unnecessary conversions later */
      return (CELL*) 0 ;
   switch (cp->type) {
      case C_DOUBLE:
         ⟨ if the *cp is an integer, find by integer value else find by string value 4a ⟩
         break ;
      case C_NOINIT:
         ap = find_by_sval(A, &null_str, create_flag, &redid) ;
         break ;
      default:
         ap = find_by_sval(A, string(cp), create_flag, &redid) ;
         break ;
   }
   return ap ? &ap->cell : (CELL *) 0 ;
}
```

See also 9b, 11a, 13a, 18b, 19a and 20a.

This code is used in 22a.

To test whether `cp->dval` is integer, we convert to the nearest integer by rounding towards zero (done by `do_to_I`) and then cast back to double. If we get the same number we started with, then `cp->dval` is integer valued.

⟨ if the `*cp` is an integer, find by integer value else find by string value 4a ⟩ ≡

```
{
   double d = cp->dval ;
   Int ival = d_to_I(d) ;
   if ((double)ival == d) {
      if (A->type == AY_SPLIT) {
         if (ival >= 1 && ival <= (int) A->size)
            return (CELL*)A->ptr+(ival-1) ;
         if (!create_flag) return (CELL*) 0 ;
         convert_split_array_to_table(A) ;
      }
      else if (A->type == AY_NULL) make_empty_table(A, AY_INT) ;
      ap = find_by_ival(A, ival, create_flag, &redid) ;
   }
   else {
      /* convert to string */
      char buff[260] ;
      STRING *sval ;
      sprintf(buff, string(CONVFMT)->str, d) ;
      sval = new_STRING(buff) ;
      ap = find_by_sval(A, sval, create_flag, &redid) ;
      free_STRING(sval) ;
   }
}
```

This code is used in 3b.

When we get to the function `find_by_ival`, the search has been reduced to lookup in a hash table by integer value.

⟨ local functions 5a ⟩≡

```
static ANODE* find_by_ival(
   ARRAY A ,
   Int ival ,
   int create_flag ,
   int *redo )
{
   DUAL_LINK *table = (DUAL_LINK*) A->ptr ;
   unsigned indx = (unsigned) ival & A->hmask ;
   ANODE *p = table[indx].ilink ; /* walks ilist */
   ANODE *q = (ANODE*) 0 ; /* trails p */
   while(1) {
      if (!p) {
          /* search failed */
          ⟨ search by string value if needed and create if needed 6a ⟩
          break ;
      }
      else if (p->ival == ival) {
         /* found it, now move to the front */
         if (!q) /* already at the front */
            return p ;
         /* delete for insertion at the front */
         q->ilink = p->ilink ;
         break ;
      }
      q = p ; p = q->ilink ;
   }
   /* insert at the front */
   p->ilink = table[indx].ilink ;
   table[indx].ilink = p ;
   return p ;
}
```

See also 7a, 8c, 14b, 15a and 16a.

This code is used in 22a.

⟨ local constants, defs and prototypes 5b ⟩+≡

```
static ANODE* find_by_ival(ARRAY, Int, int, int*);
```

See also 1c, 2a, 7b, 8a, 9a, 14a, 14c, 15b and 16b.

This code is used in 22a.

When a search by integer value fails, we have to check by string value to correctly handle the case insertion by `A["123"]` and later search as `A[123]`. This string search is necessary if and only if the `AY_STR` bit is set. An important point is that all `ANODEs` get created with a valid `sval` if `AY_STR` is set, because then creation of new nodes always occurs in a call to `find_by_sval`.

⟨ search by string value if needed and create if needed 6a ⟩ ≡

```
if (A->type & AY_STR) {
   /* need to search by string */
   char buff[256] ;
   STRING *sval ;
   sprintf(buff, INT_FMT, ival) ;
   sval = new_STRING(buff) ;
   p = find_by_sval(A, sval, create_flag, redo) ;
   if (*redo) {
      table = (DUAL_LINK*) A->ptr ;
   }
   free_STRING(sval) ;
   if (!p) return (ANODE*) 0 ;
}
else if (create_flag) {
   p = ZMALLOC(ANODE) ;
   p->sval = (STRING*) 0 ;
   p->cell.type = C_NOINIT ;
   if (++A->size > A->limit) {
      double_the_hash_table(A) ; /* changes table, may change index */
      table = (DUAL_LINK*) A->ptr ;
      indx = A->hmask & (unsigned) ival ;
   }
}
else return (ANODE*) 0 ;
p->ival = ival ;
A->type |= AY_INT ;
```

This code is used in 5a.

Searching by string value is easier because AWK arrays are really string associations. If the array does not have the AY_STR bit set, then we have to convert the array to a dual hash table with strings which is done by the function add_string_associations.

⟨ local functions 7a ⟩+≡

```
static ANODE* find_by_sval(
   ARRAY A ,
   STRING *sval ,
   int create_flag ,
   int *redo )
{
   unsigned hval = ahash(sval) ;
   char *str = sval->str ;
   DUAL_LINK *table ;
   unsigned indx ;
   ANODE *p ;  /* walks list */
   ANODE *q = (ANODE*) 0 ; /* trails p */
   if (! (A->type & AY_STR)) add_string_associations(A) ;
   table = (DUAL_LINK*) A->ptr ;
   indx = hval & A->hmask ;
   p = table[indx].slink ;
   *redo = 0 ;
   while(1) {
      if (!p)  {
         if (create_flag) {
            ⟨ create a new anode for sval 8b ⟩
            break ;
         }
         return (ANODE*) 0 ;
      }
      else if (p->hval == hval) {
         if (strcmp(p->sval->str,str) == 0 ) {
            /* found */
            if (!q) /* already at the front */
               return p ;
            else { /* delete for move to the front */
               q->slink = p->slink ;
               break ;
            }
         }
      }
      q = p ; p = q->slink ;
   }
   p->slink = table[indx].slink ;
   table[indx].slink = p ;
   return p ;
}
```

See also 5a, 8c, 14b, 15a and 16a.

This code is used in 22a.

⟨ local constants, defs and prototypes 7b ⟩+≡

```
static ANODE* find_by_sval(ARRAY, STRING*, int, int*);
```

See also 1c, 2a, 5b, 8a, 9a, 14a, 14c, 15b and 16b.

This code is used in 22a.

One `Int` value is reserved to show that the `ival` field is invalid. This works because `d_to_I` returns a value in `[-Max_Int, Max_Int]`.

⟨ local constants, defs and prototypes 8a ⟩ +≡

```
#define NOT_AN_IVALUE (-Max_Int-1)  /* usually 0x80000000 */
```

See also 1c, 2a, 5b, 7b, 9a, 14a, 14c, 15b and 16b.

This code is used in 22a.

⟨ create a new anode for `sval` 8b ⟩ ≡

```
{
   p = ZMALLOC(ANODE) ;
   p->sval = sval ;
   sval->ref_cnt++ ;
   p->ival = NOT_AN_IVALUE ;
   p->hval = hval ;
   p->cell.type = C_NOINIT ;
   if (++A->size > A->limit) {
      double_the_hash_table(A) ; /* changes table, may change index */
      table = (DUAL_LINK*) A->ptr ;
      indx = hval & A->hmask ;
      *redo = 1 ;
   }
}
```

This code is used in 7a.

On entry to `add_string_associations`, we know that the `AY_STR` bit is not set. We convert to a dual hash table, then walk all the integer lists and put each `ANODE` on a string list.

⟨ local functions 8c ⟩ +≡

```
static void add_string_associations(ARRAY A)
{
   if (A->type == AY_NULL) make_empty_table(A, AY_STR) ;
   else {
      DUAL_LINK *table ;
      int i ; /* walks table */
      ANODE *p ; /* walks ilist */
      char buff[256] ;
      if (A->type == AY_SPLIT) convert_split_array_to_table(A) ;
      table = (DUAL_LINK*) A->ptr ;
      for(i=0; (unsigned) i <= A->hmask; i++) {
         p = table[i].ilink ;
         while(p) {
            sprintf(buff, INT_FMT, p->ival) ;
            p->sval = new_STRING(buff) ;
            p->hval = ahash(p->sval) ;
            p->slink = table[A->hmask&p->hval].slink ;
            table[A->hmask&p->hval].slink = p ;
            p = p->ilink ;
         }
      }
      A->type |= AY_STR ;
   }
}
```

See also 5a, 7a, 14b, 15a and 16a.

This code is used in 22a.

⟨ local constants, defs and prototypes 9a ⟩ +≡

```
static void add_string_associations(ARRAY);
```

See also 1c, 2a, 5b, 7b, 8a, 14a, 14c, 15b and 16b.

This code is used in 22a.

3.2. Array Delete. The execution of the statement, *delete* $A[expr]$, creates a call to `array_delete(ARRAY A, CELL *cp)`. Depending on the type of `*cp`, the call is routed to `find_by_sval` or `find_by_ival`. Each of these functions leaves its return value on the front of an slist or ilist, respectively, and then it is deleted from the front of the list. The case where $A[expr]$ is on two lists, e.g., `A[12]` and `A["12"]` is checked by examining the `sval` and `ival` fields of the returned `ANODE*`.

⟨ interface functions 9b ⟩ +≡

```
void array_delete(
   ARRAY A,
   CELL *cp)
{
   ANODE *ap ;
   int redid ;
   if (A->size == 0) return ;
   switch(cp->type) {
      case C_DOUBLE :
         {
            double d = cp->dval ;
            Int ival = d_to_I(d) ;
            if ((double)ival == d) ⟨ delete by integer value and return 10a ⟩
            else { /* get the string value */
               char buff[260] ;
               STRING *sval ;
               sprintf(buff, string(CONVFMT)->str, d) ;
               sval = new_STRING(buff) ;
               ap = find_by_sval(A, sval, NO_CREATE, &redid) ;
               free_STRING(sval) ;
            }
         }
         break ;
      case C_NOINIT :
         ap = find_by_sval(A, &null_str, NO_CREATE, &redid) ;
         break ;
      default :
         ap = find_by_sval(A, string(cp), NO_CREATE, &redid) ;
         break ;
   }
   if (ap) { /* remove from the front of the slist */
      DUAL_LINK *table = (DUAL_LINK*) A->ptr ;
      table[ap->hval & A->hmask].slink = ap->slink ;
      ⟨ if ival is valid, remove ap from its ilist 10c ⟩
      free_STRING(ap->sval) ;
      cell_destroy(&ap->cell) ;
      ZFREE(ap) ;
      ⟨ decrement A->size 10d ⟩
   }
}
```

See also 3b, 11a, 13a, 18b, 19a and 20a.

This code is used in 22a.

⟨ delete by integer value and return 10a ⟩ ≡

```
{
   if (A->type == AY_SPLIT)
     {
      if (ival >=1 && ival <= (int) A->size)
          convert_split_array_to_table(A) ;
      else return ; /* ival not in range */
     }
   ap = find_by_ival(A, ival, NO_CREATE, &redid) ;
   if (ap) { /* remove from the front of the ilist */
      DUAL_LINK *table = (DUAL_LINK*) A->ptr ;
      table[(unsigned) ap->ival & A->hmask].ilink = ap->ilink ;
      ⟨ if sval is valid, remove ap from its slist 10b ⟩
      cell_destroy(&ap->cell) ;
      ZFREE(ap) ;
      ⟨ decrement A->size 10d ⟩
   }
   return ;
}
```

This code is used in 9b.

Even though we found a node by searching an ilist it might also be on an slist and vice-versa.

⟨ if sval is valid, remove ap from its slist 10b ⟩ ≡

```
if (ap->sval) {
   ANODE *p, *q = 0 ;
   unsigned indx = (unsigned) ap->hval & A->hmask ;
   p = table[indx].slink ;
   while(p != ap) { q = p ; p = q->slink ; }
   if (q) q->slink = p->slink ;
   else table[indx].slink = p->slink ;
   free_STRING(ap->sval) ;
}
```

This code is used in 10a.

⟨ if ival is valid, remove ap from its ilist 10c ⟩ ≡

```
if (ap->ival != NOT_AN_IVALUE) {
   ANODE *p, *q = 0 ;
   unsigned indx = (unsigned) ap->ival & A->hmask ;
   p = table[indx].ilink ;
   while(p != ap) { q = p ; p = q->ilink ; }
   if (q) q->ilink = p->ilink ;
   else table[indx].ilink = p->ilink ;
}
```

This code is used in 9b.

When the size of a hash table drops below a certain value, it might be profitable to shrink the hash table. Currently we don't do this, because our guess is that it would be a waste of time for most AWK applications. However, we do convert an array to AY_NULL when the size goes to zero which would resize a large hash table that had been completely cleared by successive deletions.

⟨ decrement A->size 10d ⟩ ≡

```
if (--A->size == 0) array_clear(A) ;
```

This code is used in 9b and 10a.

3.3. Building an Array with Split. A simple operation is to create an array with the `AWK` primitive `split`. The code that performs `split` puts the pieces in the global buffer `split_buff`. The call `array_load(A, cnt)` moves the `cnt` elements from `split_buff` to `A`. This is the only way an array of type `AY_SPLIT` is created.

⟨ interface functions 11a ⟩ +≡
```
void array_load(
   ARRAY A,
   size_t cnt)
{
   CELL *cells ; /* storage for A[1..cnt] */
   size_t i ;  /* index into cells[] */
   ⟨ clean up the existing array and prepare an empty split array 12a ⟩
   cells = (CELL*) A->ptr ;
   A->size = cnt ;
   ⟨ if cnt exceeds MAX_SPLIT, load from overflow list and adjust cnt 11b ⟩
   for(i=0;i < cnt; i++) {
      cells[i].type = C_MBSTRN ;
      cells[i].ptr = split_buff[i] ;
   }
}
```

See also 3b, 9b, 13a, 18b, 19a and 20a.

This code is used in 22a.

When `cnt > MAX_SPLIT`, `split_buff` was not big enough to hold everything so the overflow went on the `split_ov_list`. The elements from `MAX_SPLIT+1` to `cnt` get loaded into `cells[MAX_SPLIT..cnt-1]` from this list.

⟨ if cnt exceeds MAX_SPLIT, load from overflow list and adjust cnt 11b ⟩ ≡
```
if (cnt > MAX_SPLIT) {
   SPLIT_OV *p = split_ov_list ;
   SPLIT_OV *q ;
   split_ov_list = (SPLIT_OV*) 0 ;
   i = MAX_SPLIT ;
   while( p ) {
      cells[i].type = C_MBSTRN ;
      cells[i].ptr = (PTR) p->sval ;
      q = p ; p = q->link ; ZFREE(q) ;
      i++ ;
   }
   cnt = MAX_SPLIT ;
}
```

This code is used in 11a.

If the array `A` is a split array and big enough then we reuse it, otherwise we need to allocate a new split array. When we allocate a block of `CELL`s for a split array, we round up to a multiple of 4.

⟨ clean up the existing array and prepare an empty split array 12a ⟩ ≡

```
if (A->type != AY_SPLIT || A->limit < (unsigned) cnt) {
   array_clear(A) ;
   A->limit = (unsigned) ( (cnt & (size_t) ~3) + 4 ) ;
   A->ptr = zmalloc(A->limit*sizeof(CELL)) ;
   A->type = AY_SPLIT ;
}
else
{
   for(i=0; (unsigned) i < A->size; i++)
       cell_destroy((CELL*)A->ptr + i) ;
}
```

This code is used in 11a.

3.4. Array Clear. The function `array_clear(ARRAY A)` converts `A` to type `AY_NULL` and frees all storage used by `A` except for the `struct array` itself. This function gets called in two contexts: (1) when an array local to a user function goes out of scope and (2) execution of the `AWK` statement, `delete A`.

⟨ interface functions 13a ⟩ +≡

```
void array_clear(ARRAY A)
{
   unsigned i ;
   ANODE *p, *q ;
   if (A->type == AY_SPLIT) {
      for(i = 0; i < A->size; i++)
          cell_destroy((CELL*)A->ptr+i) ;
      zfree(A->ptr, A->limit * sizeof(CELL)) ;
   }
   else if (A->type & AY_STR) {
      DUAL_LINK *table = (DUAL_LINK*) A->ptr ;
      for(i=0; (unsigned) i <= A->hmask; i++) {
         p = table[i].slink ;
         while(p) {
            q = p ; p = q->slink ;
            free_STRING(q->sval) ;
            cell_destroy(&q->cell) ;
            ZFREE(q) ;
         }
      }
      zfree(A->ptr, (A->hmask+1)*sizeof(DUAL_LINK)) ;
   }
   else if (A->type & AY_INT) {
      DUAL_LINK *table = (DUAL_LINK*) A->ptr ;
      for(i=0; (unsigned) i <= A->hmask; i++) {
         p = table[i].ilink ;
         while(p) {
            q = p ; p = q->ilink ;
            cell_destroy(&q->cell) ;
            ZFREE(q) ;
         }
      }
      zfree(A->ptr, (A->hmask+1)*sizeof(DUAL_LINK)) ;
   }
   memset(A, 0, sizeof(*A)) ;
}
```

See also 3b, 9b, 11a, 18b, 19a and 20a.

This code is used in 22a.

3.5. Constructor and Conversions. Arrays are always created as empty arrays of type `AY_NULL`. Global arrays are never destroyed although they can go empty or have their type change by conversion. The only constructor function is a macro.

⟨ array typedefs and `#defines` 13b ⟩ +≡

```
#define new_ARRAY()  ((ARRAY)memset(ZMALLOC(struct array),0,sizeof(struct array)))
```

See also 1a, 1b and 3a.

This code is used in 21d.

Hash tables only get constructed by conversion. This happens in two ways. The function `make_empty_table` converts an empty array of type `AY_NULL` to an empty hash table. The number of lists in the table is a power of 2 determined by the constant `STARTING_HMASK`. The limit size of the table is determined by the constant `MAX_AVE_LIST_LENGTH` which is the largest average size of the hash lists that we are willing to tolerate before enlarging the table. When `A->size` exceeds `A->limit`, the hash table grows in size by doubling the number of lists. `A->limit` is then reset to `MAX_AVE_LIST_LENGTH` times `A->hmask+1`.

⟨ local constants, defs and prototypes 14a ⟩ +≡

```
#define STARTING_HMASK    63  /* 2^6-1, must have form 2^n-1 */
#define MAX_AVE_LIST_LENGTH   12
#define hmask_to_limit(x) (((x)+1)*MAX_AVE_LIST_LENGTH)
#define ahash(sval) hash2((sval)->str, (sval)->len)
```

See also 1c, 2a, 5b, 7b, 8a, 9a, 14c, 15b and 16b.

This code is used in 22a.

⟨ local functions 14b ⟩ +≡

```
static void make_empty_table(
   ARRAY A ,
   int type ) /* AY_INT or AY_STR */
{
   size_t sz = (STARTING_HMASK+1)*sizeof(DUAL_LINK) ;
   A->type = (short) type ;
   A->hmask = STARTING_HMASK ;
   A->limit = hmask_to_limit(STARTING_HMASK) ;
   A->ptr = memset(zmalloc(sz), 0, sz) ;
}
```

See also 5a, 7a, 8c, 15a and 16a.

This code is used in 22a.

⟨ local constants, defs and prototypes 14c ⟩ +≡

```
static void make_empty_table(ARRAY, int);
```

See also 1c, 2a, 5b, 7b, 8a, 9a, 14a, 15b and 16b.

This code is used in 22a.

The other way a hash table gets constructed is when a split array is converted to a hash table of type `AY_INT`.

⟨ local functions 15a ⟩ +≡
```
static void convert_split_array_to_table(ARRAY A)
{
   CELL *cells = (CELL*) A->ptr ;
   unsigned i ; /* walks cells */
   DUAL_LINK *table ;
   unsigned j ; /* walks table */
   size_t entry_limit = A->limit ;
   ⟨ determine the size of the hash table and allocate 15c ⟩
   /* insert each cells[i] in the new hash table on an ilist */
   for(i=0, j=1; i < A->size; i++) {
      ANODE *p = ZMALLOC(ANODE) ;
      p->sval = (STRING*) 0 ;
      p->ival = (Int) (i + 1) ;
      p->cell = cells[i] ;
      p->ilink = table[j].ilink ;
      table[j].ilink = p ;
      j++ ; j &= A->hmask ;
   }
   A->type = AY_INT ;
   zfree(cells, entry_limit*sizeof(CELL)) ;
}
```

See also 5a, 7a, 8c, 14b and 16a.

This code is used in 22a.

⟨ local constants, defs and prototypes 15b ⟩ +≡
```
static void convert_split_array_to_table(ARRAY);
```

See also 1c, 2a, 5b, 7b, 8a, 9a, 14a, 14c and 16b.

This code is used in 22a.

To determine the size of the table, we set the initial size to `STARTING_HMASK+1` and then double the size until `A->size <= A->limit`.

⟨ determine the size of the hash table and allocate 15c ⟩ ≡
```
A->hmask = STARTING_HMASK ;
A->limit = hmask_to_limit(STARTING_HMASK) ;
while(A->size > A->limit) {
   A->hmask = (A->hmask<<1) + 1 ; /* double the size */
   A->limit = hmask_to_limit(A->hmask) ;
}
{
   size_t sz = (A->hmask+1)*sizeof(DUAL_LINK) ;
   A->ptr = memset(zmalloc(sz), 0, sz) ;
   table = (DUAL_LINK*) A->ptr ;
}
```

This code is used in 15a.

3.6. Doubling the Size of a Hash Table. The whole point of making the table size a power of two is to facilitate resizing the table. If the table size is 2^n and h is the hash key, then $h \bmod 2^n$ is the hash chain index which can be calculated with bit-wise and, $h \,\&\, (2^n - 1)$. When the table size doubles, the new bit-mask has one more bit turned on. Elements of an old hash chain whose hash value have this bit turned on get moved to a new chain. Elements with this bit turned off stay on the same chain. On average only half the old chain moves to the new chain. If the old chain is at $table[i],\ 0 \le i < 2^n$, then the elements that move, all move to the new chain at $table[i + 2^n]$.

⟨ local functions 16a ⟩ +≡
```
static void double_the_hash_table(ARRAY A)
{
   unsigned old_hmask = A->hmask ;
   unsigned new_hmask = (old_hmask<<1)+1 ;
   DUAL_LINK *table ;
   ⟨ allocate the new hash table 16c ⟩
   ⟨ if the old table has string lists, move about half the string nodes 16d ⟩
   ⟨ if the old table has integer lists, move about half the integer nodes 17b ⟩
   A->hmask = new_hmask ;
   A->limit = hmask_to_limit(new_hmask) ;
}
```

See also 5a, 7a, 8c, 14b and 15a.

This code is used in 22a.

⟨ local constants, defs and prototypes 16b ⟩ +≡
```
static void double_the_hash_table(ARRAY);
```

See also 1c, 2a, 5b, 7b, 8a, 9a, 14a, 14c and 15b.

This code is used in 22a.

⟨ allocate the new hash table 16c ⟩ ≡
```
A->ptr = zrealloc(A->ptr, (old_hmask+1)*sizeof(DUAL_LINK),
                          (new_hmask+1)*sizeof(DUAL_LINK)) ;
table = (DUAL_LINK*) A->ptr ;
/* zero out the new part which is the back half */
memset(&table[old_hmask+1], 0, (old_hmask+1)*sizeof(DUAL_LINK)) ;
```

This code is used in 16a.

⟨ if the old table has string lists, move about half the string nodes 16d ⟩ ≡
```
if (A->type & AY_STR) {
   unsigned i ; /* index to old lists */
   unsigned j ; /* index to new lists */
   ANODE *p ; /* walks an old list */
   ANODE *q ; /* trails p for deletion */
   ANODE *tail ; /* builds new list from the back */
   ANODE dummy0, dummy1 ;
   for(i=0, j=old_hmask+1; i <= old_hmask; i++, j++)
      ⟨ walk one old string list, creating one new string list 17a ⟩
}
```

This code is used in 16a.

As we walk an old string list with pointer `p`, the expression `p->hval & new_hmask` takes one of two values. If it is equal to `p->hval & old_hmask` (which equals `i`), then the node stays otherwise it gets moved to a new string list at `j`. The new string list preserves order so that the positions of the move-to-the-front heuristic are preserved. Nodes moving to the new list are appended at pointer `tail`. The `ANODEs`, `dummy0` and `dummy1`, are sentinels that remove special handling of boundary conditions.

⟨ walk one old string list, creating one new string list 17a ⟩ ≡

```
{
   q = &dummy0 ;
   q->slink = p = table[i].slink ;
   tail = &dummy1 ;
   while (p) {
      if ((p->hval & new_hmask) != (unsigned) i) { /* move it */
         q->slink = p->slink ;
         tail = tail->slink = p ;
      }
      else q = p ;
      p = q->slink ;
   }
   table[i].slink = dummy0.slink ;
   tail->slink = (ANODE*) 0 ;
   table[j].slink = dummy1.slink ;
}
```

This code is used in 16d.

The doubling of the integer lists is exactly the same except that `slink` is replaced by `ilink` and `hval` is replaced by `ival`.

⟨ if the old table has integer lists, move about half the integer nodes 17b ⟩ ≡

```
if (A->type & AY_INT) {
   unsigned i ; /* index to old lists */
   unsigned j ; /* index to new lists */
   ANODE *p ; /* walks an old list */
   ANODE *q ; /* trails p for deletion */
   ANODE *tail ; /* builds new list from the back */
   ANODE dummy0, dummy1 ;
   for(i=0, j=old_hmask+1; i <= old_hmask; i++, j++)
      ⟨ walk one old integer list, creating one new integer list 18a ⟩
}
```

This code is used in 16a.

⟨ walk one old integer list, creating one new integer list 18a ⟩ ≡

```
{
   q = &dummy0 ;
   q->ilink = p = table[i].ilink ;
   tail = &dummy1 ;
   while (p) {
      if (((unsigned) p->ival & new_hmask) != i) { /* move it */
         q->ilink = p->ilink ;
         tail = tail->ilink = p ;
      }
      else q = p ;
      p = q->ilink ;
   }
   table[i].ilink = dummy0.ilink ;
   tail->ilink = (ANODE*) 0 ;
   table[j].ilink = dummy1.ilink ;
}
```

This code is used in 17b.

Initializing Array Loops Our mechanism for dealing with execution of the statement,

`for(i in A) {` *statements* `}`

is simple. We allocate a vector of `STRING*` of size, `A->size`. Each element of the vector is a string key for `A`. Note that if the `AY_STR` bit of `A` is not set, then `A` has to be converted to a string hash table, because the index `i` walks string indices.

To execute the loop, the only state that needs to be saved is the address of `i` and an index into the vector of string keys. Since nothing about `A` is saved as state, the user program can do anything to `A` inside the body of the loop, even `delete A`, and the loop still works. Essentially, we have traded data space (the string vector) in exchange for implementation simplicity. On a 32-bit system, each `ANODE` is 36 bytes, so the extra memory needed for the array loop is 11% more than the memory consumed by the `ANODEs` of the array. Note that the large size of the `ANODEs` is indicative of our whole design which pays data space for integer lookup speed and algorithm simplicity.

The only aspect of array loops that occurs in `array.c` is construction of the string vector. The rest of the implementation is in the file `execute.c`.

⟨ interface functions 18b ⟩ +≡

```
static int string_compare(
   const void *l,
   const void *r)
{
   STRING*const * a = (STRING *const *) l;
   STRING*const * b = (STRING *const *) r;

   return strcmp((*a)->str, (*b)->str);
}
```

See also 3b, 9b, 11a, 13a, 19a and 20a.

This code is used in 22a.

⟨ interface functions 19a ⟩ +≡

```
STRING** array_loop_vector(
   ARRAY A,
   size_t *sizep)
{
   STRING** ret ;
   *sizep = A->size ;
   if (A->size > 0) {
      if (!(A->type & AY_STR)) add_string_associations(A) ;
      ret = (STRING**) zmalloc(A->size*sizeof(STRING*)) ;
      ⟨ for each ANODE in A, put one string in ret 19b ⟩
      if (getenv("WHINY_USERS") != NULL)        /* gawk compability */
        qsort(ret, A->size, sizeof(STRING*), string_compare);
      return ret ;
   }
   return (STRING**) 0 ;
}
```

See also 3b, 9b, 11a, 13a, 18b and 20a.

This code is used in 22a.

As we walk over the hash table ANODEs, putting each sval in ret, we need to increment each reference count. The user of the return value is responsible for these new reference counts.

⟨ for each ANODE in A, put one string in ret 19b ⟩ ≡

```
{
   int r = 0 ; /* indexes ret */
   DUAL_LINK* table = (DUAL_LINK*) A->ptr ;
   int i ; /* indexes table */
   ANODE *p ; /* walks slists */
   for(i=0; (unsigned) i <= A->hmask; i++) {
      for(p = table[i].slink; p ; p = p->slink) {
         ret[r++] = p->sval ;
         p->sval->ref_cnt++ ;
      }
   }
}
```

This code is used in 19a.

3.7. Concatenating Array Indices. In AWK, an array expression `A[i,j]` is equivalent to the expression `A[i SUBSEP j]`, i.e., the index is the concatenation of the three elements `i`, `SUBSEP` and `j`. This is performed by the function `array_cat`. On entry, `sp` points at the top of a stack of `CELLs`. `Cnt` cells are popped off the stack and concatenated together separated by `SUBSEP` and the result is pushed back on the stack. On entry, the first multi-index is in `sp[1-cnt]` and the last is in `sp[0]`. The return value is the new stack top. (The stack is the run-time evaluation stack. This operation really has nothing to do with array structure, so logically this code belongs in `execute.c`, but remains here for historical reasons.)

⟨ interface functions 20a ⟩ +≡

```
CELL *array_cat(
   CELL *sp,
   int cnt)
{
   CELL *p ;  /* walks the eval stack */
   CELL subsep ;  /* local copy of SUBSEP */
   ⟨ subsep parts 20b ⟩
   size_t total_len ;  /* length of cat'ed expression */
   CELL *top ;   /* value of sp at entry */
   char *target ;  /* build cat'ed char* here */
   STRING *sval ;  /* build cat'ed STRING here */
   ⟨ get subsep and compute parts 20c ⟩
   ⟨ set top and return value of sp 20d ⟩
   ⟨ cast cells to string and compute total_len 21a ⟩
   ⟨ build the cat'ed STRING in sval 21b ⟩
   ⟨ cleanup, set sp and return 21c ⟩
}
```

See also 3b, 9b, 11a, 13a, 18b and 19a.

This code is used in 22a.

We make a copy of `SUBSEP` which we can cast to string in the unlikely event the user has assigned a number to `SUBSEP`.

⟨ subsep parts 20b ⟩ ≡

```
size_t subsep_len ; /* string length of subsep_str */
char *subsep_str ;
```

This code is used in 20a.

⟨ get subsep and compute parts 20c ⟩ ≡

```
cellcpy(&subsep, SUBSEP) ;
if ( subsep.type < C_STRING ) cast1_to_s(&subsep) ;
subsep_len = string(&subsep)->len ;
subsep_str = string(&subsep)->str ;
```

This code is used in 20a.

Set `sp` and `top` so the cells to concatenate are inclusively between `sp` and `top`.

⟨ set `top` and return value of `sp` 20d ⟩ ≡

```
assert(cnt > 0);

top = sp ; sp -= (cnt-1) ;
```

This code is used in 20a.

The `total_len` is the sum of the lengths of the `cnt` strings and the `cnt-1` copies of `subsep`.

⟨ cast cells to string and compute `total_len` 21a ⟩≡
```
total_len = ((size_t) (cnt-1)) * subsep_len ;
for(p = sp ; p <= top ; p++) {
   if ( p->type < C_STRING ) cast1_to_s(p) ;
   total_len += string(p)->len ;
}
```
This code is used in 20a.

⟨ build the cat'ed `STRING` in `sval` 21b ⟩≡
```
sval = new_STRING0(total_len) ;
target = sval->str ;
for(p = sp ; p < top ; p++) {
   memcpy(target, string(p)->str, string(p)->len) ;
   target += string(p)->len ;
   memcpy(target, subsep_str, subsep_len) ;
   target += subsep_len ;
}
/* now p == top */
memcpy(target, string(p)->str, string(p)->len) ;
```
This code is used in 20a.

The return value is `sp` and it is already set correctly. We just need to free the strings and set the contents of `sp`.

⟨ cleanup, set `sp` and return 21c ⟩≡
```
for(p = sp; p <= top ; p++) free_STRING(string(p)) ;
free_STRING(string(&subsep)) ;
/* set contents of sp , sp->type > C_STRING is possible so reset */
sp->type = C_STRING ;
sp->ptr = (PTR) sval ;
return sp ;
```
This code is used in 20a.

4. Source Files.

⟨ `"array.h"` 21d ⟩≡
```
/* array.h */
```
⟨ blurb 22b ⟩
```
#ifndef ARRAY_H
#define ARRAY_H 1

#include "nstd.h"
#include "types.h"
```
⟨ array typedefs and `#defines` 1a, ... ⟩
⟨ interface prototypes 2b ⟩
```
#endif /* ARRAY_H */
```

⟨ "array.c" 22a ⟩≡
```
/* array.c */
⟨ blurb 22b ⟩
#include "mawk.h"
#include "symtype.h"
#include "memory.h"
#include "field.h"
#include "bi_vars.h"
⟨ local constants, defs and prototypes 1c, ... ⟩
⟨ interface functions 3b, ... ⟩
⟨ local functions 5a, ... ⟩
```

⟨ blurb 22b ⟩≡
```
/*
$MawkId: array.w,v 1.15 2010/12/10 17:00:00 tom Exp $

copyright 2009,2010, Thomas E. Dickey
copyright 1991-1996, Michael D. Brennan

This is a source file for mawk, an implementation of
the AWK programming language.

Mawk is distributed without warranty under the terms of
the GNU General Public License, version 2, 1991.

array.c and array.h were generated with the commands

   notangle -R'"array.c"' array.w > array.c
   notangle -R'"array.h"' array.w > array.h

Notangle is part of Norman Ramsey's noweb literate programming package
available from CTAN(ftp.shsu.edu).

It's easiest to read or modify this file by working with array.w.
*/
```

This code is used in 21d and 22a.

5. Identifier Index. Underlined code chunks are identifier definitions; other chunks are identifier uses.