Mawk Arrays

rcs-version `1.6` `2016/07/20`

Table of Contents

1. Introduction.

This is the source and documentation for the `mawk` implementation of AWK arrays. Arrays in AWK are associations of strings to awk scalar values.

2. 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 `ptr` field depends on the `type` field.

⟨ array typedefs and `#defines` 1a ⟩ ≡

```
    typedef struct array {
       void* ptr ;  /* What this points to depends on the type */
       size_t size ; /* number of elts in the table */
       int type ;  /* values in AY_NULL .. AY_SPLIT */
    } *ARRAY ;
```

See also 1b, 1c, 4b, 7b and 9a.

This code is used in 21a.

By AWK language specification, there is only one kind of array, but internally `mawk` has four kinds of arrays. These are distinguished by the `type` field in the structure. The types are,

- `AY_NULL` The array is empty. The `size` and `ptr` fields are zero.
- `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 `CELL`s. The number of `CELL`s is the `size` field. The address of `A[i]` is `(CELL*)A->ptr+i-1`.
- `AY_STR` The `ptr` field points at an associative map from `STRING*` to `CELL*`. If `s` is an awk string, then this associative map holds the memory for `A[s]`.
- `AY_INT` The `ptr` field points at an associative map from `integer` to `CELL*`. If `d` is an awk numeric expression and of integer value, then this associative map holds the memory `A[d]`. An array of this type must always be accessed with an integer key. If it is accessed with a string key, then the array is converted to type `AY_STR`.

⟨ array typedefs and `#defines` 1b ⟩ +≡

```
    enum {
       AY_NULL = 0,
       AY_SPLIT,
       AY_STR,
       AY_INT
    } ;
```

See also 1a, 1c, 4b, 7b and 9a.

This code is used in 21a.

2.1. Construction.

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` 1c ⟩ +≡

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

See also 1a, 1b, 4b, 7b and 9a.

This code is used in 21a.

2.2. Internal Tables. String (`AY_STR`) and integer (`AY_INT`) arrays are implemented with internal tables of type `ITable`. The operations on `ITable`s are,

`make_empty_itable()` returns a pointer to an empty `ITable`.

`itable_free(it)` empties the table and then frees all memory associated with the table. The table is then totally gone.

The main operations performed on a table are done by `itable_find()`.

⟨ local constants, defines and prototypes 2a ⟩≡
```
typedef struct itable ITable ;
static CELL* itable_find(ITable* it, STRING* sval, int64_t ival,
                         int flag, size_t* sizep) ;
```
See also 3c and 11a.
This code is used in 21b.

The parameters are,

`it` find in table `*it`.

`sval` if `sval` is non-zero, find by string key `sval`. If `A` is the array represented internally by `*it`, then the return value is a pointer to `A[sval]`.

`ival` if `sval` is zero, find by integer key `ival`. If `A` is the array represented internally by `*it`, then the return value is a pointer to `A[ival]`.

`flag` can take three values: `NO_CREATE`, `CREATE` and `DELETE_`. If `flag` is `NO_CREATE` and the key is not in the table, then zero is returned. If `flag` is `CREATE` and the key is not in the table, then create a new element in the table of `CELL` type, `C_NOINIT`. If `flag` is `DELETE_` and the key is found, then that element of the table is deleted and zero is returned.

`sizep` if creation or deletion changes the number of elements in the table, then `*sizep` is set to the new size (+1 the old size on creation or -1 the old size on deletion).

2.3. Array Type Conversions. By specification of the AWK language, an array is an association of string value to AWK scalar, `A[3] = 1` means `A["3"] = 1`. So in theory, there is only one array type, `AY_STR`. In practice, `mawk` uses `AY_SPLIT` and `AY_INT` for faster performance. This works fine as long as `A[3]` is always accessed as `A[3]`, but while `A[3]=1; print A["3"]` is weird, it is also correct and must print `1`. Handling this example correctly requires array `A` be converted from `AY_INT` to `AY_STR`. There are three conversions,

`AY_SPLIT` to `AY_INT`

`AY_SPLIT` to `AY_STR`

`AY_INT` to `AY_STR`

Once converted, an array is never converted back to its original form.

This is `AY_INT` to `AY_STR`.

⟨ local functions 2b ⟩≡
```
static void array_int_to_str(ARRAY A)
{
    ITable* ht = (ITable*)A->ptr ;
    itable_convert_i_to_s(ht) ;
    A->type = AY_STR ;
}
```
See also 3a, 3b and 10b.
This code is used in 21b.

This is AY_SPLIT to AY_INT. Each element of the split array is inserted into an integer array.

⟨local functions 3a⟩+≡
```
static void array_split_to_int(ARRAY A)
{
    ITable* tb = make_empty_itable() ;
    unsigned i ;
    CELL* cells = (CELL*)A->ptr ;
    for(i=1; i <= A->size; i++) {
        size_t unused ;
        CELL* cp = itable_find(tb,0, i, CREATE, &unused) ;
        *cp = cells[i-1] ; /* no ref cnt adjustment needed */
    }
    zfree(cells, sizeof(CELL) * A->size) ;
    A->type = AY_INT ;
    A->ptr = tb ;
    /* A->size stayed the same */
}
```
See also 2b, 3b and 10b.

This code is used in 21b.

This is AY_SPLIT to AY_STR. Each element of the split array is inserted into a string array. This involves converting each key to string with sprintf().

⟨local functions 3b⟩+≡
```
static void array_split_to_str(ARRAY A)
{
    ITable* tb = make_empty_itable() ;
    unsigned i ;
    CELL* cells = (CELL*)A->ptr ;
    for(i=1; i <= A->size; i++) {
        size_t unused ;
        char buffer[128] ;
        STRING* sval ;
        CELL* cp ;
        sprintf(buffer, "%u" , i) ;
        sval = new_STRING(buffer) ;
        cp = itable_find(tb, sval, 0, CREATE, &unused) ;
        *cp = cells[i-1] ; /* no ref cnt adjustment needed */
        free_STRING(sval) ;
    }
    zfree(cells, sizeof(CELL) * A->size) ;
    A->type = AY_STR ;
    A->ptr = tb ;
    /* A->size stayed the same */
}
```
See also 2b, 3a and 10b.

This code is used in 21b.

⟨local constants, defines and prototypes 3c⟩+≡
```
static void array_int_to_str(ARRAY) ;
static void array_split_to_int(ARRAY) ;
static void array_split_to_str(ARRAY) ;
```
See also 2a and 11a.

This code is used in 21b.

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 `C_NOINIT`.

`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 moved into `A` from an anonymous buffer with `transfer_to_array()` which is declared in `split.h`.

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

`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 the concatenation expression, `i SUBSEP j`.

⟨ interface prototypes 4a ⟩ ≡
```
    CELL* array_find(ARRAY, CELL*, int);
    void  array_delete(ARRAY, CELL*);
    void  array_load(ARRAY, size_t);
    void  array_clear(ARRAY);
    CELL* array_cat(CELL*, int);
```
See also 12a.
This code is used in 21a.

There are also interface functions for looping over the indices of an array that are detailed later.

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` 4b ⟩ +≡
```
    #define NO_CREATE  0
    #define CREATE     1
```
See also 1a, 1b, 1c, 7b and 9a.
This code is used in 21a.

How `Array_find` works depends on the type of the array. If the type of the key, `*cp`, matches the array type, then lookup is straightforward; otherwise, the array type needs conversion.

```
⟨interface functions 5a⟩≡
    CELL* array_find(ARRAY A, CELL *cp, int create_flag)
    {
        CELL key ; /* a copy of *cp */
        if (A->size == 0 && !create_flag) {
            return 0 ;
        }
        cellcpy(&key,cp) ;
    reswitch:
        switch(A->type) {
            case AY_NULL:
                ⟨make a new integer or string array 6b⟩
            case AY_SPLIT:
                ⟨find in a split array 6a⟩
            case AY_STR:
                {
                    CELL* ret ;
                    cast1_to_s(&key) ;
                    ret = itable_find((ITable*)(A->ptr),string(&key), 0,
                              create_flag, &A->size) ;
                    free_STRING(string(&key)) ;
                    return ret ;
                }
            case AY_INT:
                {
                    if (key.type != C_DOUBLE || !is_int_double(key.dval)) {
                        array_int_to_str(A) ;
                        goto reswitch ;
                    }
                    /* the expected case */
                    return itable_find((ITable*)A->ptr, 0, (int64_t)key.dval,
                               create_flag, &A->size) ;
                }
        }
        /* not reached */
        return 0 ;
    }
```

See also 7a, 8a, 8b, 10a, 11b, 11c and 12b.
This code is used in 21b.

If `1 <= key <= A->size`, then lookup is a simple array reference, else a conversion is needed.

⟨find in a split array 6a⟩≡
```
    {
        if (key.type != C_DOUBLE || !is_int_double(key.dval)) {
            array_split_to_str(A) ;
            goto reswitch ;
        }
        if (key.dval < 1.0 || key.dval > (double) A->size) {
            if (create_flag) {
                array_split_to_int(A) ;
                goto reswitch ;
            }
            else  return 0 ;
        }
        else {
            /* the expected case */
            CELL* cells = (CELL*)A->ptr ;
            unsigned d = (unsigned) key.dval ;
            return &cells[d-1] ;
        }
    }
```
This code is used in 5a.

One element is added to an empty table. The table type is `AY_INT` or `AY_STR` depending on the type of the key.

⟨make a new integer or string array 6b⟩≡
```
    {
        ITable* tb = make_empty_itable() ;
        A->ptr = tb ;
        A->size = 0 ;
        if (key.type != C_DOUBLE || !is_int_double(key.dval)) {
            A->type = AY_STR ;
            goto reswitch ;
        }
        A->type = AY_INT ;
        return itable_find(tb, 0, (int64_t) key.dval, CREATE, &A->size) ;
    }
```
This code is used in 5a.

3.2. Array Delete. The execution of the statement, `delete A[expr]`, creates a call to `array_delete(A,cp)`, where `cp` points at the `CELL` holding `expr`. Depending on the type of array `A` and the type of `*cp`, the array may undergo type conversion similar to that with `array_find(A,cp)`. After that, it is a call to `itable_find()` for deletion. If deletion makes the size zero, the array `type` becomes `AY_NULL`.

⟨ interface functions 7a ⟩ +≡
```
    void array_delete(ARRAY A, CELL* cp)
    {
        CELL key ; /* copy of *cp */
        if (A->type == AY_NULL) return ;

        cellcpy(&key,cp) ;
    reswitch:
        switch(A->type) {
            case AY_STR:
                cast1_to_s(&key) ;
                itable_find((ITable*)A->ptr, string(&key), 0, DELETE_, &A->size) ;
                free_STRING(string(&key)) ;
                break ;
            case AY_INT:
                if (key.type != C_DOUBLE || !is_int_double(key.dval)) {
                    array_int_to_str(A) ;
                    goto reswitch ;
                }
                itable_find((ITable*)A->ptr, 0, (int64_t) key.dval,
                            DELETE_, &A->size) ;
                break ;
            case AY_SPLIT:
                if (key.type != C_DOUBLE || !is_int_double(key.dval)) {
                    array_split_to_str(A) ;
                }
                else if (key.dval < 1.0 || key.dval > (double)A->size) {
                    /* not in the array so nothing to do */
                    return ;
                }
                else {
                    array_split_to_int(A) ;
                }
                goto reswitch ;
        }
        if (A->size == 0) array_clear(A) ;
    }
```
See also 5a, 8a, 8b, 10a, 11b, 11c and 12b.

This code is used in 21b.

⟨ array typedefs and `#defines` 7b ⟩ +≡
```
    #define DELETE_   2
```
See also 1a, 1b, 1c, 4b and 9a.

This code is used in 21a.

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 an anonymous buffer. `array_load(A, cnt)` moves the `cnt` elements from the anonymous buffer into `A`. This is the only way an array of type `AY_SPLIT` is created.

⟨interface functions 8a⟩+≡
```
void array_load(ARRAY A, size_t cnt)
{
    array_clear(A) ;
    if (cnt > 0) {
        CELL* cells = (CELL*)zmalloc(sizeof(CELL) * cnt) ;
        memset(cells, 0, sizeof(CELL) * cnt) ;
        A->size = cnt ;
        A->type = AY_SPLIT ;
        A->ptr = cells ;
        transfer_to_array(cells, cnt) ;
    }
}
```
See also 5a, 7a, 8b, 10a, 11b, 11c and 12b.
This code is used in 21b.

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 four contexts: (1) when an array local to a user function goes out of scope, (2) execution of the `AWK` statement, `delete A`, (3) execution of the `AWK` statement, `delete A[expr]`, deletes the last element, and (4) when an existing array is used by `split()`.

⟨interface functions 8b⟩+≡
```
void array_clear(ARRAY A)
{
    if (A->type == AY_NULL) return ;

    if (A->type == AY_SPLIT) {
        unsigned i ;
        for(i = 0; i < A->size; i++) {
            cell_destroy((CELL*)A->ptr+i) ;
        }
        zfree(A->ptr, A->size * sizeof(CELL)) ;
    }
    else {
        itable_free((ITable*)A->ptr) ;
    }

    A->ptr = 0 ;
    A->size = 0 ;
    A->type = AY_NULL ;
}
```
See also 5a, 7a, 8a, 10a, 11b, 11c and 12b.
This code is used in 21b.

3.5. Array Loops. The loop over the array indices in,

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

is controlled by an `ALoop`. The fields in an `ALoop` are,

`type` is the type of array `A`.

`size` is the size of `A`.

`next` is an index into the indices of `A`.

`cp` is a pointer to the `CELL` address of loop variable `i`.

`link` is pointer used to put the `ALoop` object in a stack, which handles nested loops.

Since the `ALoop` object holds all state necessary to run the loop, the user program can do anything to `A` inside the body of the loop, even `delete A`, and the loop still works.

⟨array typedefs and `#defines` 9a⟩+≡

```
typedef struct aloop {
    struct aloop* link ;
    int type ;  /* AY_NULL .. AY_INT */
    unsigned size ;
    unsigned next ;
    CELL* cp ;
    union {
        STRING** sval ;  /* for AY_STR */
        int64_t*  ival ; /* for AY_INT */
    } ptr ;
} ALoop ;
```

See also 1a, 1b, 1c, 4b and 7b.
This code is used in 21a.

The interface functions are,

`make_aloop(A,cp)` constructs an `ALoop` for looping on `(i in A)`; `cp` is a pointer to the `CELL` address of `i`.

`aloop_free(al)` destructs and frees all memory held by `ALoop`, `al`.

`aloop_next(al)` puts the next array index into variable, `i`, at `al->cp` and returns `1`. If the loop is complete, `i` is unchanged and the return is `0`.

Note, that the `ITable` interface provides a hook for getting the array indices into a vector.

⟨interface functions 10a⟩+≡
```
ALoop* make_aloop(ARRAY A, CELL* cp)
{
    ALoop* al = (ALoop*)zmalloc(sizeof(ALoop)) ;
    al->type = A->type ;
    al->size = A->size ;
    al->next = 0 ;
    al->cp = cp ;
    al->link = 0 ;
    if (al->type == AY_INT) {
        al->ptr.ival = itable_i_vector((ITable*)(A->ptr)) ;
#ifdef QSORT
        qsort(al->ptr.ival, al->size, sizeof(int64_t), i_compare) ;
#endif
    }
    else if (al->type == AY_STR) {
        al->ptr.sval = itable_s_vector((ITable*)(A->ptr)) ;
#ifdef QSORT
        qsort(al->ptr.sval, al->size, sizeof(STRING*), s_compare) ;
#endif
    }
    return al ;
}
```

See also 5a, 7a, 8a, 8b, 11b, 11c and 12b.

This code is used in 21b.

⟨local functions 10b⟩+≡
```
#ifdef QSORT
static int i_compare(const void* l, const void* r)
{
    const int64_t* il = l ;
    const int64_t* ir = r ;
    return *il - *ir ;
}

static int s_compare(const void* l, const void* r)
{
    STRING* const* sl = l ;
    STRING* const* sr = r ;
    return STRING_cmp(*sl,*sr) ;
}
#endif
```

See also 2b, 3a and 3b.

This code is used in 21b.

⟨ local constants, defines and prototypes 11a ⟩ +≡
```
#ifdef QSORT
static int i_compare(const void*, const void*) ;
static int s_compare(const void*, const void*) ;
#endif
```
See also 2a and 3c.

This code is used in 21b.

⟨ interface functions 11b ⟩ +≡
```
void aloop_free(ALoop* al)
{
    if (al->type == AY_INT) {
        zfree(al->ptr.ival, sizeof(int64_t) * al->size) ;
    }
    else if (al->type == AY_STR) {
        unsigned i ;
        for(i=0; i < al->size; i++) {
            free_STRING(al->ptr.sval[i]) ;
        }
        zfree(al->ptr.sval, sizeof(STRING*) * al->size) ;
    }
    zfree(al, sizeof(ALoop)) ;
}
```
See also 5a, 7a, 8a, 8b, 10a, 11c and 12b.

This code is used in 21b.

How the loop is indexed depends on the type.

⟨ interface functions 11c ⟩ +≡
```
int aloop_next(ALoop* al)
{
    if (al->next >= al->size) return 0 ;
    cell_destroy(al->cp) ;
    switch(al->type) {
        case AY_SPLIT:
            al->cp->type = C_DOUBLE ;
            al->cp->dval = (double) (al->next+1) ;
            break ;
        case AY_INT:
            al->cp->type = C_DOUBLE ;
            al->cp->dval = (double) al->ptr.ival[al->next] ;
            break ;
        case AY_STR:
            al->cp->type = C_STRING ;
            al->cp->ptr = STRING_dup(al->ptr.sval[al->next]) ;
            break ;
    }
    al->next++ ;
    return 1 ;
}
```
See also 5a, 7a, 8a, 8b, 10a, 11b and 12b.

This code is used in 21b.

⟨interface prototypes 12a⟩+≡
```
ALoop* make_aloop(ARRAY, CELL*) ;
void aloop_free(ALoop*) ;
int aloop_next(ALoop*) ;
```
See also 4a.
This code is used in 21a.

3.6. 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 12b⟩+≡
```
CELL *array_cat(
   CELL *sp,
   int cnt)
{
   CELL *p ;  /* walks the eval stack */
   CELL subsep ;  /* local copy of SUBSEP */
   ⟨subsep parts 12c⟩
   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 12d⟩
   ⟨set top and return value of sp 13a⟩
   ⟨cast cells to string and compute total_len 13b⟩
   ⟨build the cat'ed STRING in sval 13c⟩
   ⟨cleanup, set sp and return 13d⟩
}
```
See also 5a, 7a, 8a, 8b, 10a, 11b and 11c.
This code is used in 21b.

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 12c⟩≡
```
size_t subsep_len ; /* string length of subsep_str */
char *subsep_str ;
```
This code is used in 12b.

⟨get subsep and compute parts 12d⟩≡
```
{
    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 12b.

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

⟨set top and return value of sp 13a⟩≡
```
{
    top = sp ;
    sp -= (cnt-1) ;
}
```
This code is used in 12b.

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 13b⟩≡
```
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 12b.

⟨build the cat'ed STRING in sval 13c⟩≡
```
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 12b.

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 13d⟩≡
```
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 12b.

4. Hash Table.

Up to this point, the internal tables (`ITable`) have only provided a functional interface. Here is the implementation; it is a hash table. A different design such as red-black tree could be used to provide the same interface.

The hash table design 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.

4.1. Data Structure.

Each element of the table is an `HNODE`. If the keys are integer (`AY_INT`), then the key is `ival`. If the keys are `STRING*` (`AY_STR`), then the key is `key`. Note that an `ITable` holds no information about array type.

The fields are,

`link` connects `HNODE`s into buckets of singly linked list.

`key` the node is keyed on `STRING*` value if `key` is not zero.

`ival` if `key` is zero, the node is keyed on this integer value.

`cell` the value looked up by either `key` or `ival`.

⟨hash table declarations and data 14a⟩≡

```
    typedef struct hnode {
        struct hnode* link ;
        int64_t ival ;
        STRING* key ;
        CELL cell ;
    } HNODE ;
```

See also 14b, 15b, 17b, 18a, 19b and 20c.

This code is used in 21b.

Each bucket of a hash table is a linked list of `HNODE`s. The number of buckets is always a power of 2. If the number of buckets is `2^n`, then the `hmask` is `2^n-1`. For `HNODE* p`, `p` is on the linked list starting at `buckets[table->hmask & p->ival`.

⟨hash table declarations and data 14b⟩+≡

```
    #define INIT_HMASK   255
    #define MAX_AVE_BUCKET_SIZE   4
    struct itable {
        unsigned hmask ;
        unsigned size ;
        unsigned limit ;
        HNODE** buckets ;
    } ;
```

See also 14a, 15b, 17b, 18a, 19b and 20c.

This code is used in 21b.

⟨hash table functions 14c⟩≡

```
    static ITable* make_empty_itable()
    {
        ITable* ret = ZMALLOC(ITable) ;
        ret->hmask = INIT_HMASK ;
        ret->size = 0 ;
        ret->limit = MAX_AVE_BUCKET_SIZE * (INIT_HMASK+1) ;
        ret->buckets = make_buckets(INIT_HMASK+1) ;
        return ret ;
    }
```

See also 15a, 15c, 16b, 17c, 18b, 20a and 20b.

This code is used in 21b.

⟨ hash table functions 15a ⟩ +≡
```
static HNODE** make_buckets(unsigned cnt)
{
    HNODE** bks = (HNODE**)emalloc(cnt * sizeof(HNODE*)) ;
    memset(bks, 0, cnt * sizeof(HNODE*)) ;
    return bks ;
}
```
See also 14c, 15c, 16b, 17c, 18b, 20a and 20b.
This code is used in 21b.

⟨ hash table declarations and data 15b ⟩ +≡
```
static HNODE** make_buckets(unsigned) ;
static ITable* make_empty_itable(void) ;
```
See also 14a, 14b, 17b, 18a, 19b and 20c.
This code is used in 21b.

4.2. Find, Create and Delete.

⟨ hash table functions 15c ⟩ +≡
```
CELL* itable_find(ITable* htable, STRING* sval, int64_t ival,
                  int flag, size_t* szp)
{
    int64_t hash = sval ? hash2(sval->str,sval->len) : ival ;
    unsigned idx = (unsigned) hash & htable->hmask ;
    HNODE* q = 0 ;
    HNODE* p = htable->buckets[idx] ;
    while(p) {
        if (hash == p->ival && (!sval || STRING_eq(sval,p->key))) {
            /* found */
            if (flag == DELETE_) {
                ⟨ delete p and return 0 16a ⟩
            }
            if (q) {
                /* move to front */
                q->link = p->link ;
                p->link = htable->buckets[idx] ;
                htable->buckets[idx] = p ;
            }  /* else at front */
            return &p->cell ;
        }
        q = p ;
        p = p->link ;
    }
    /* not found */
    if (flag == CREATE) {
        ⟨ create a new node at p and insert in htable at buckets[idx] 17a ⟩
        return &p->cell ;
    }
    return 0 ;
}
```
See also 14c, 15a, 16b, 17c, 18b, 20a and 20b.
This code is used in 21b.

⟨ delete p and return 0 16a ⟩≡
```
{
    htable->size-- ;
    *szp = htable->size ;
    if (q) {
        q->link = p->link ;
    }
    else {
        htable->buckets[idx] = p->link ;
    }
    hnode_free(p) ;
    return 0 ;
}
```
This code is used in 15c.

Function, `hnode_free(p)`, frees all memory used by `HNODE`, `*p`. Function, `htable_free(ht)`, frees all memory used by `ITable`, `*ht`.

⟨ hash table functions 16b ⟩+≡
```
static void hnode_free(HNODE* p)
{
    if (p->key) free_STRING(p->key) ;
    cell_destroy(&p->cell) ;
    zfree(p, sizeof(HNODE)) ;
}
static void itable_free(ITable* ht)
{
    unsigned i ;
    unsigned size = ht->size ;
    for(i=0; size > 0; i++) {
        HNODE* p = ht->buckets[i] ;
        while(p) {
            HNODE* q = p ;
            p = p->link ;
            hnode_free(q) ;
            size-- ;
        }
    }
    free(ht->buckets) ;
    zfree(ht, sizeof(ITable)) ;
}
```
See also 14c, 15a, 15c, 17c, 18b, 20a and 20b.
This code is used in 21b.

⟨create a new node at `p` and insert in `htable` at `buckets[idx]` 17a⟩≡

```
    {
        p = ZMALLOC(HNODE) ;
        p->key = sval ? STRING_dup(sval) : 0 ;
        p->ival = hash ;
        p->cell.type = C_NOINIT ;
        p->link = htable->buckets[idx] ;
        htable->buckets[idx] = p ;
        htable->size++ ;
        *szp = htable->size ;
        if (htable->size > htable->limit) {
            double_num_buckets(htable) ;
        }
    }
```

This code is used in 15c.

⟨hash table declarations and data 17b⟩+≡

```
    static void hnode_free(HNODE*) ;
    static void itable_free(ITable*) ;
```

See also 14a, 14b, 15b, 18a, 19b and 20c.
This code is used in 21b.

Function, `itable_convert_i_to_s(ht)`, takes as input a hash table keyed on integers (`ival`), converts each integer key to a string key via `sprintf()`, and rebuilds the table on the string keys.

⟨hash table functions 17c⟩+≡

```
    static void itable_convert_i_to_s(ITable* ht)
    {
        char buffer[256] ;
        unsigned hmask = ht->hmask ;
        HNODE** new_buckets = make_buckets(hmask+1) ;
        unsigned i ;  /* walks old buckets */
        unsigned j ;  /* index into new_buckets */
        unsigned cnt = ht->size ;  /* number of nodes to convert */
        for(i=0; cnt > 0; i++) {
            HNODE* p = ht->buckets[i] ;
            while(p) {
                HNODE* q = p ;
                p = p->link ;
                sprintf(buffer, LDFMT, q->ival) ;
                q->key = new_STRING(buffer) ;
                q->ival = hash(buffer) ;
                j = q->ival & hmask ;
                q->link = new_buckets[j] ;
                new_buckets[j] = q ;
                cnt-- ;
            }
        }
        free(ht->buckets) ;
        ht->buckets = new_buckets ;
    }
```

See also 14c, 15a, 15c, 16b, 18b, 20a and 20b.
This code is used in 21b.

⟨ hash table declarations and data 18a ⟩ +≡
```
    static void itable_convert_i_to_s(ITable*) ;
```
See also 14a, 14b, 15b, 17b, 19b and 20c.
This code is used in 21b.

4.3. Doubling the Number of Hash Buckets. The whole point of making the number of buckets a power of two is to facilitate resizing. If the number of buckets is `2^n` and `h` is the hash key, then `h & 2^n-1` is the hash bucket index. When the number of buckets doubles, the new bit-mask has one more bit turned on. Elements of an old hash bucket, whose hash value have this bit turned on, get moved to a new bucket. Elements, with this bit turned off, stay in the same bucket. On average only half the old bucket moves to the new bucket. If the old bucket is at `buckets[i]` for `i < 2^n`, then the elements that move, all move to the new bucket at `buckets[i+2^n]`.

⟨ hash table functions 18b ⟩ +≡
```
    static void double_num_buckets(ITable* htable)
    {
       unsigned old_hmask = htable->hmask ;
       unsigned new_hmask = (old_hmask<<1)+1 ;
       HNODE** buckets = htable->buckets ;
       buckets = (HNODE**)erealloc(buckets, sizeof(HNODE*) * (new_hmask+1)) ;
       ⟨ move about half the HNODEs 19a ⟩
       htable->hmask = new_hmask ;
       htable->limit = (new_hmask+1) * MAX_AVE_BUCKET_SIZE ;
       htable->buckets = buckets ;
    }
```
See also 14c, 15a, 15c, 16b, 17c, 20a and 20b.
This code is used in 21b.

⟨ move about half the HNODEs 19a ⟩≡

```
{
    HNODE* p ;  /* walks bucket[i] */
    HNODE* q ;  /* trails p */
    HNODE* tail ; /* builds bucket[j] from the back */
    HNODE p0 ;  /* sentinel */
    HNODE t0 ;  /* sentinel */
    unsigned i, j ;

    for(i=0, j=old_hmask+1; i < old_hmask+1; i++, j++) {
        p = buckets[i] ;
        q = &p0 ;
        q->link = p ;
        tail = &t0 ;
        tail->link = 0 ;

        while(p) {
            if (p->ival & (old_hmask+1)) {
                /* it moves */
                q->link = p->link ;
                tail = tail->link = p ;
                tail->link = 0 ;
                p = q->link ;
            }
            else {
                q = p ;
                p = p->link ;
            }
        }
        buckets[i] = p0.link ;
        buckets[j] = t0.link ;
    }
}
```

This code is used in 18b.

⟨ hash table declarations and data 19b ⟩+≡

```
static void double_num_buckets(ITable*) ;
```

See also 14a, 14b, 15b, 17b, 18a and 20c.

This code is used in 21b.

4.4. Loop Vectors. Creating index vectors for array loops walks over all the table nodes placing each lookup key in a vector.

⟨ hash table functions 20a ⟩ +≡
```
static int64_t*  itable_i_vector(ITable* it)
{
    int64_t* ret = (int64_t*)zmalloc(sizeof(int64_t)*it->size) ;
    unsigned r = 0 ;
    unsigned i ;
    for(i=0; r < it->size; i++) {
        HNODE* p = it->buckets[i] ;
        while(p) {
            ret[r++] = p->ival ;
            p = p->link ;
        }
    }
    return ret ;
}
```
See also 14c, 15a, 15c, 16b, 17c, 18b and 20b.
This code is used in 21b.

⟨ hash table functions 20b ⟩ +≡
```
static STRING**  itable_s_vector(ITable* it)
{
    STRING** ret = (STRING**)zmalloc(sizeof(STRING*)*it->size) ;
    unsigned r = 0 ;
    unsigned i ;
    for(i=0; r < it->size; i++) {
        HNODE* p = it->buckets[i] ;
        while(p) {
            ret[r++] = STRING_dup(p->key) ;
            p = p->link ;
        }
    }
    return ret ;
}
```
See also 14c, 15a, 15c, 16b, 17c, 18b and 20a.
This code is used in 21b.

⟨ hash table declarations and data 20c ⟩ +≡
```
static int64_t* itable_i_vector(ITable*) ;
static STRING** itable_s_vector(ITable*) ;
```
See also 14a, 14b, 15b, 17b, 18a and 19b.
This code is used in 21b.

5. Source Files.

```
⟨"array.h" 21a⟩≡
    /* array.h */
    ⟨blurb 21c⟩
    #ifndef ARRAY_H
    #define ARRAY_H 1

    #include "types.h"

    ⟨array typedefs and #defines 1a, ...⟩
    ⟨interface prototypes 4a, ...⟩
    #endif /* ARRAY_H */
```

```
⟨"array.c" 21b⟩≡
    /* array.c */
    ⟨blurb 21c⟩
    #include "mawk.h"
    #include "int.h"
    #include "symtype.h"
    #include "memory.h"
    #include "split.h"
    #include "field.h"
    #include "bi_vars.h"
    ⟨local constants, defines and prototypes 2a, ...⟩
    ⟨hash table declarations and data 14a, ...⟩
    ⟨interface functions 5a, ...⟩
    ⟨local functions 2b, ...⟩
    ⟨hash table functions 14c, ...⟩
```

```
⟨blurb 21c⟩≡
    /*
    copyright 1991-1996,2014-2016 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 3, 2007.

    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.
    Noweb home page: http://www.cs.tufts.edu/~nr/noweb/

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

This code is used in 21a and 21b.

6. Identifier Index.

Underlined code chunks are identifier definitions; other chunks are identifier uses.