Dynamic memory in C

- Abstract data types
- Arrays that grow and shrink
  - realloc
  - memmove
- getline function

Abstract Data Types

- So far, we have described basic data types, all the standard C statements, operators and expressions, functions, and function prototypes.
- We want to introduce the concept of modularization
- Before there were object-oriented languages like Java and C++, users of imperative languages used **abstract data types** (ADT):
  - an abstract data type is a set of operations which access a collection of stored data
  - in Java and C++ this idea is called **encapsulation**

Abstract Data Types

- Since ANSI compilers support separate compilation of source modules, we can use abstract data types and function prototypes to **simulate modules**:
  - this is simply for convenience
  - a C compiler does not force us to use separate files
  - allows us to implement the "one declaration – one definition" rule

Abstract Data Types (2)

- For module "mod" there are two files
- **Interface module**: named "mod.h" contains function prototypes, public type definitions, constants, and when necessary declarations for global variables. Interface modules are also called header files.
  - Interface modules are accessed using the #include C preprocessor directive
- **Implementation module**: named "mod.c" contains the implementation of functions declared in the interface module.

Arrays that grow and shrink

- All of our C programs using arrays up to now have been static in size
  - Assignment specifications state the largest input size.
  - Memory is allocated for these arrays from the C compiler and run time
  - We never need to manage this memory.
- Arrays are a very handy data structure
  - Easy to index and access (O(1) operations)
  - Contiguous block of memory can be exploited by other functions (qsort, memcpy, memset).
- Therefore we would like to keep the convenience of arrays but also obtain the benefits of dynamic memory
  - … and do so without having to write more complex structures like lists, heaps, etc.

Nameval array

- Suppose we wish to maintain an array of <name, value> pairs
  - Name is a string
  - Value is an integer
- We want to add new items to our array as they arrive
- If there is not enough room in the array, we want to grow it.
- To support this we'll keep the array's size and items-to-date associated with the array via a struct.
  - Note use of "typedef"

```
typedef struct Nameval Nameval;
struct Nameval {
    char   *name;
    int     value;
};
```

```
struct Nvtab {
    int   nval;
    int   max;
    Nameval *nameval;
} nvtab;

enum { NVINIT = 1, NVGROW = 2 };
```

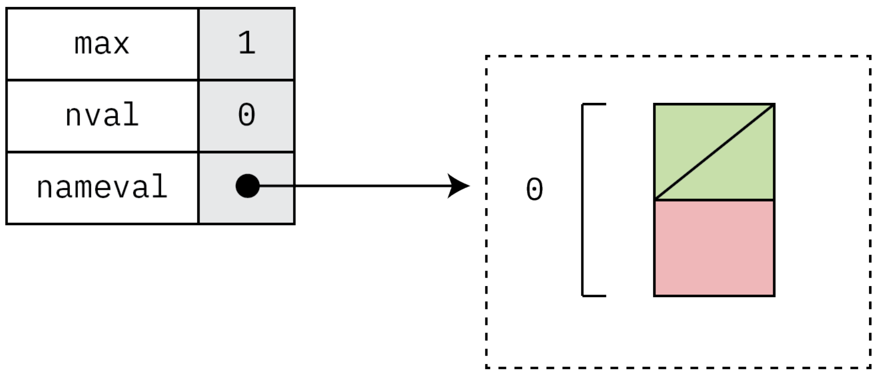
Creating a new nameval

```
Nameval *new_nameval(char *name, int value)
{
    Nameval *temp;

    temp = (Nameval *)malloc(sizeof(Nameval));
    if (temp == NULL) {
        fprintf(stderr, "Error mallocing a Nameval");
        exit(1);
    }

    /* temp->name === (*temp).name */
    temp->name = (char *)malloc((strlen(name)+1) * sizeof(char));
    if (temp->name == NULL) {
        fprintf(stderr, "Error mallocing a memory for string");
        exit(1);
    }
    strncpy(temp->name, name, strlen(name)+1);

    temp->value = value;

    return temp;
}
```

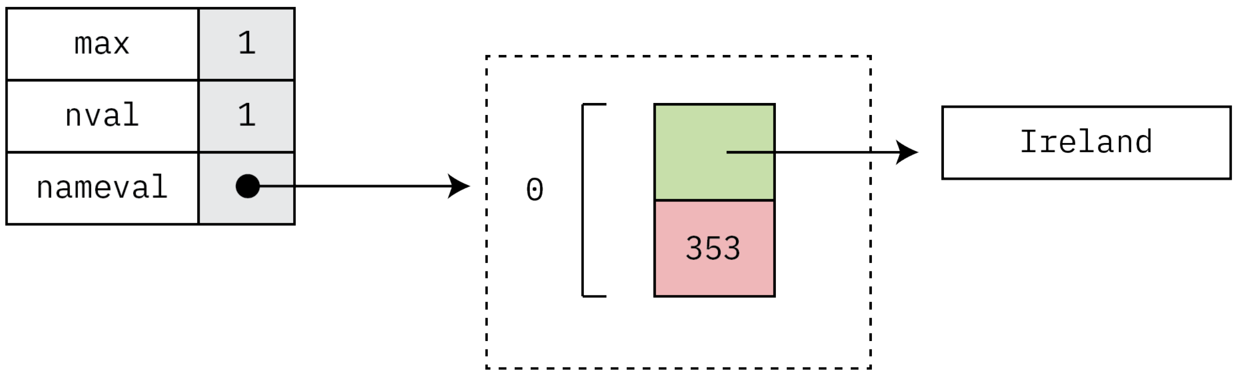
Initial value of **nvtab** variable

max	0
nval	0
nameval	

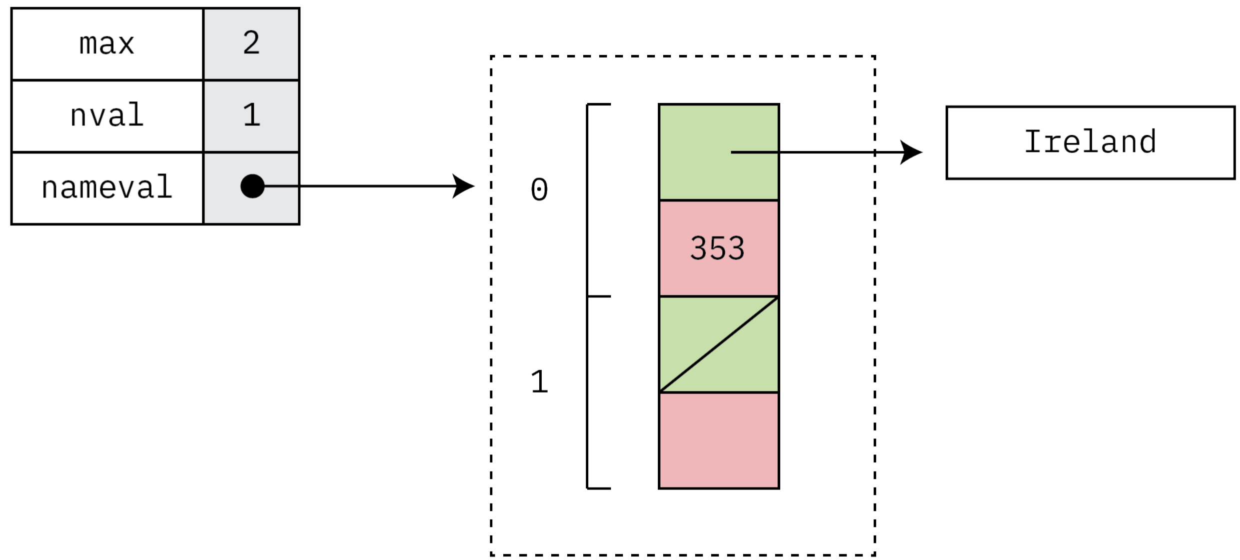
Adding ("Ireland", 353): after array in heap is allocated, but before assigning actual value



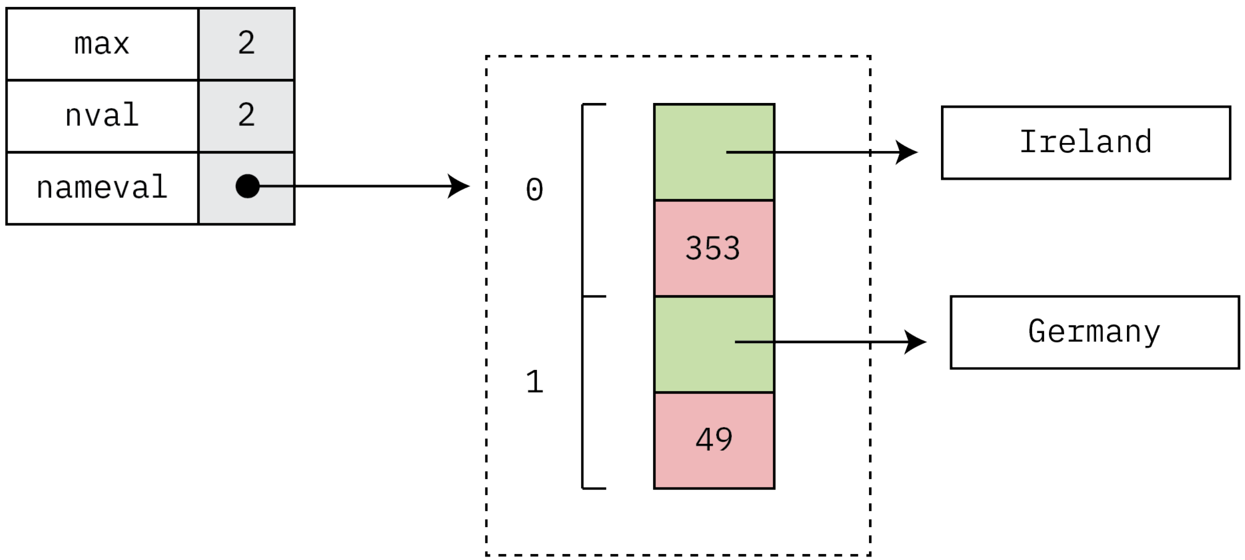
Adding ("Ireland", 353): after array in heap is allocated, and after assigning the actual value



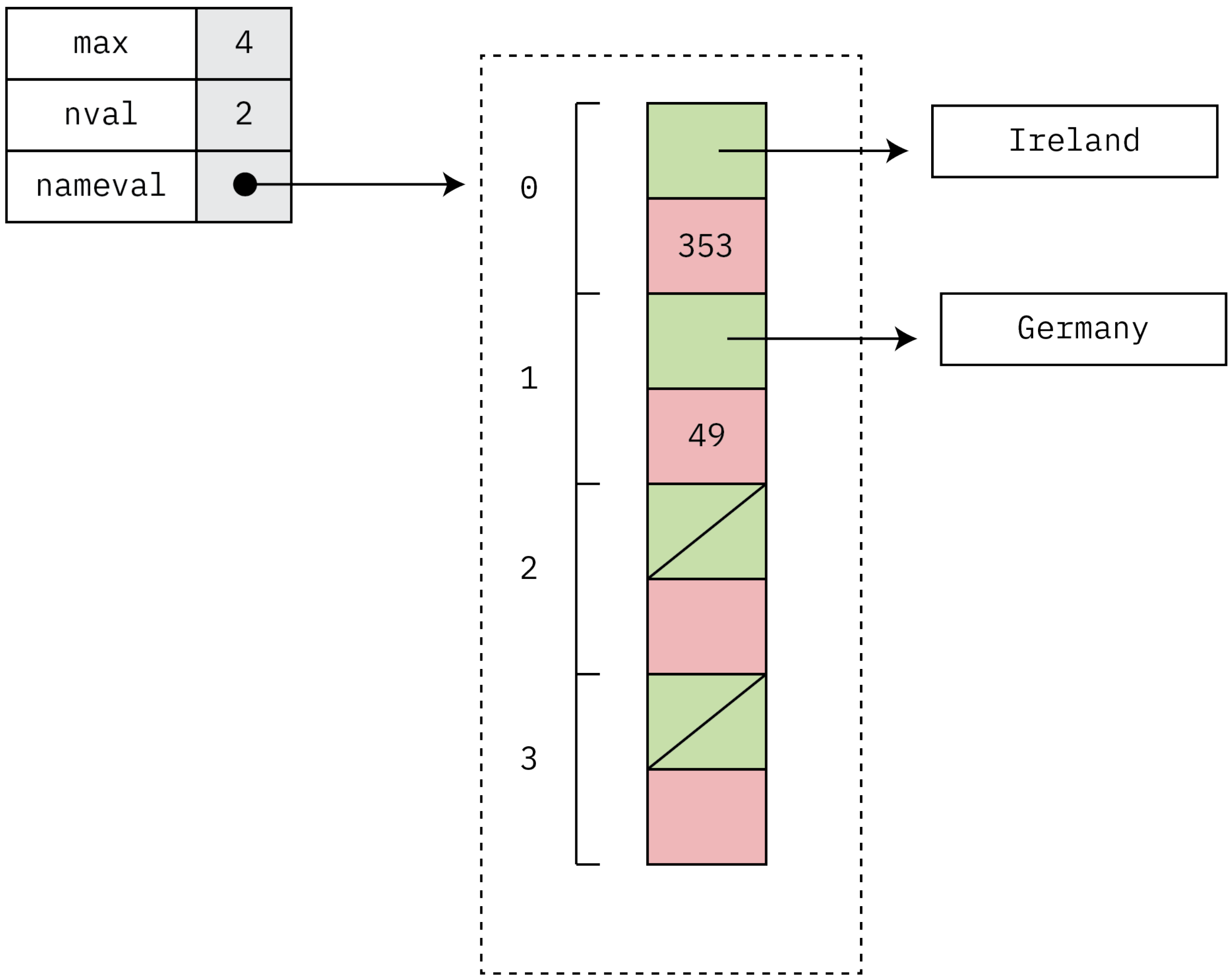
Adding ("Germany", 49): after array grows, but before assigning the actual value



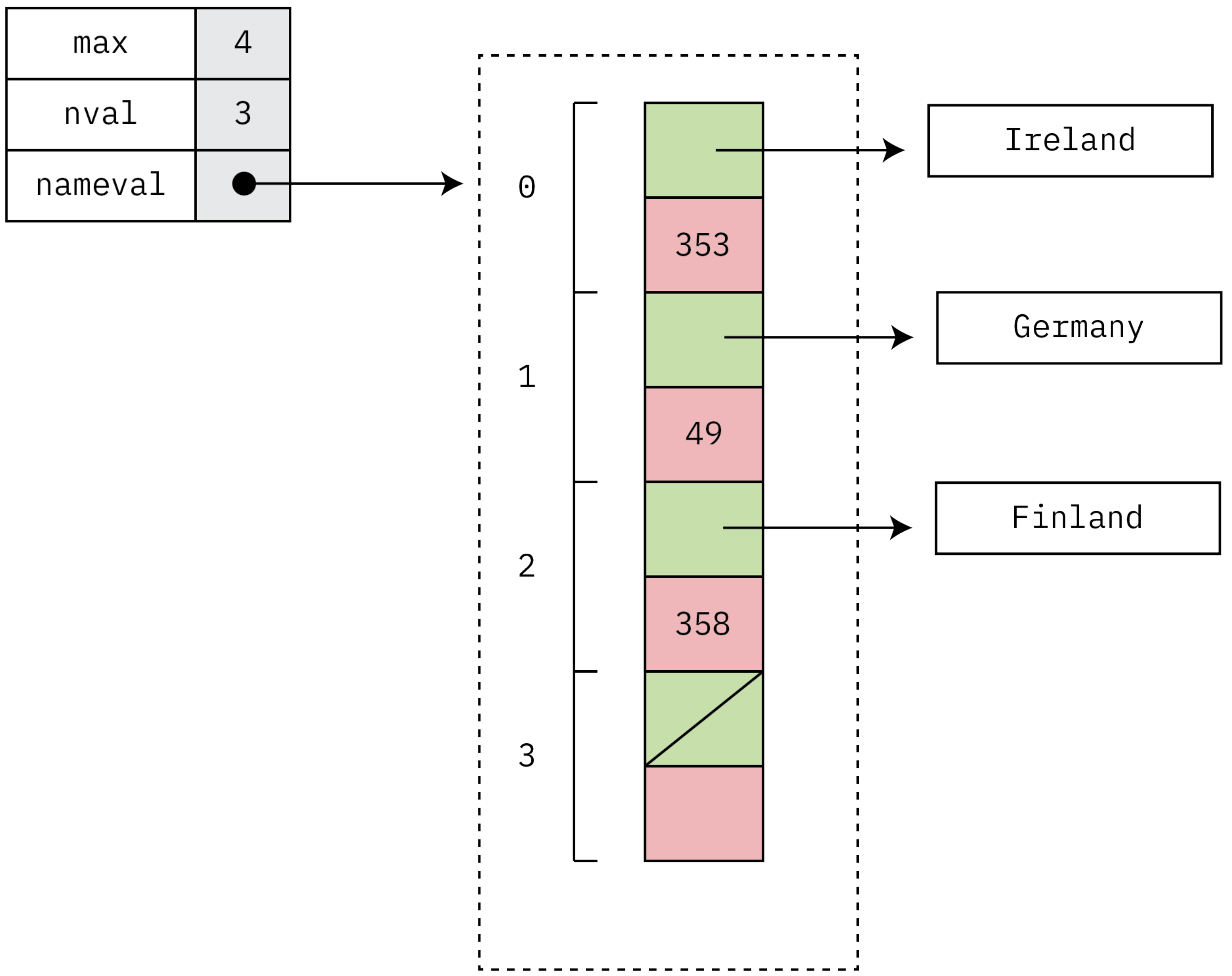
Adding ("Germany", 49): after array grows, and after assigning the actual value



Adding ("Finland", 358): after array grows, but before assigning the actual value



Adding ("Finland", 358): after array grows, and after assigning the actual value



... and notice that the next call to additem() will not result in the array needing to grow before assignment

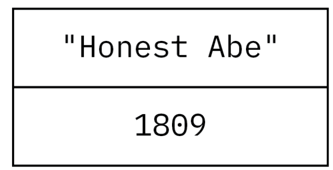
addname

```
int addname(Nameval newname)
{
    Nameval *nvp;

    if (nvtab.nameval == NULL) { /* first use of array */
        nvtab.nameval =
            (Nameval *) malloc(NVINIT * sizeof(Nameval));
        if (nvtab.nameval == NULL) { return -1; }
        nvtab.max = NVINIT;
        nvtab.nval = 0;
    } else if (nvtab.nval >= nvtab.max) {
        nvp = (Nameval *) realloc(nvtab.nameval,
            (NVGROW * nvtab.max) * sizeof(Nameval));
        if (nvp == NULL) { return -1; }
        nvtab.max = NVGROW * nvtab.max;
        nvtab.nameval = nvp;
    }
    nvtab.nameval[nvtab.nval] = newname;
    return nvtab.nval++;
}
```

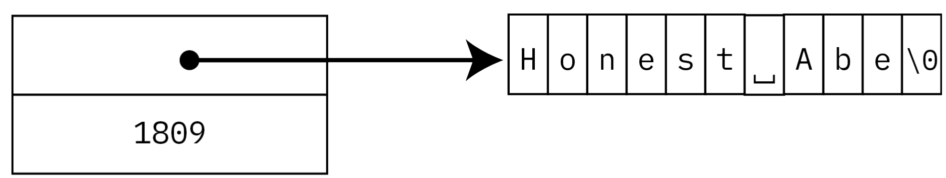
Deleting a name

- Arrays are contiguous…
  - Yet we may sometimes want to remove elements that are within the array
  - That is, neither at the start or end
- This can be tricky:
  - We need to decide what to do with the resulting gap in the array.
  - If element order doesn't matter: just swap last item in array with gap
  - If element order does matter (i.e., must be preserved), the we must move all the elements beyond the gap by one position

Reminder: This...

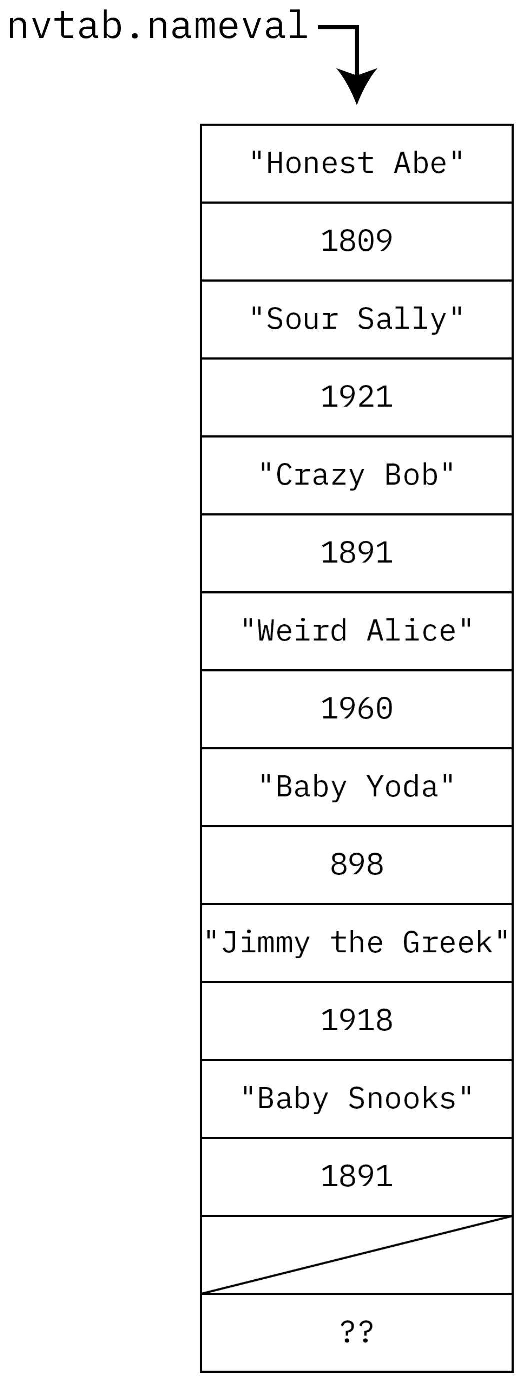


... is a shorthand way of representing this:



delname

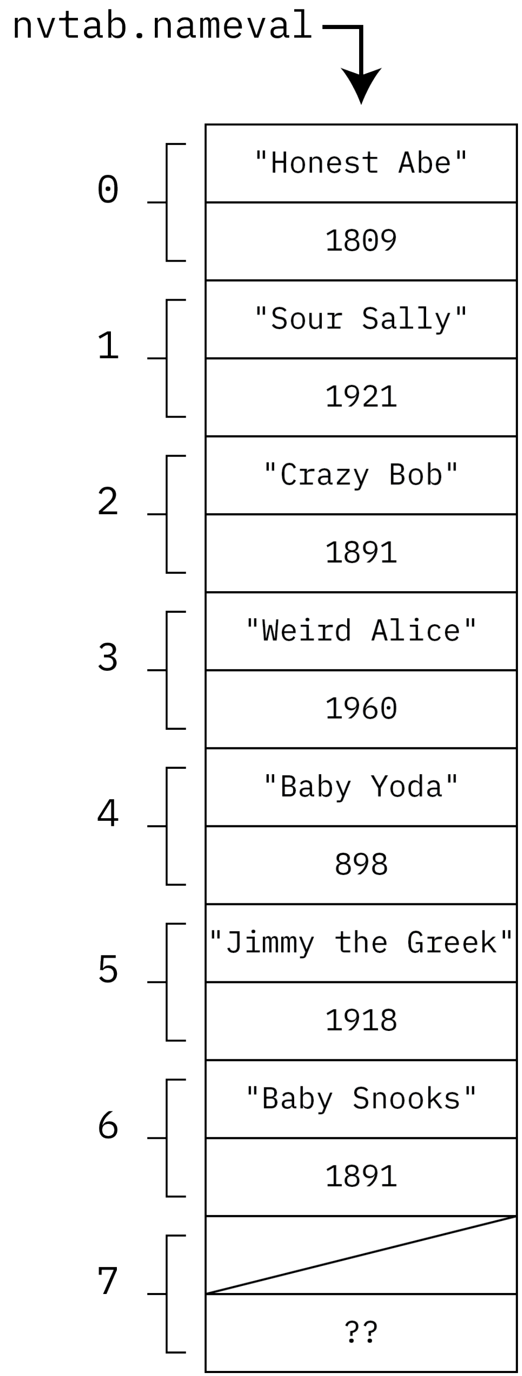
```
int delname (char *name)
{
    int i;

    for (i = 0; i < nvtab.nval; i++) {
        if (strcmp(nvtab.nameval[i].name, name) == 0) {
            memmove(nvtab.nameval + i, nvtab.nameval + i + 1,
                (nvtab.nval-(i+1)) * sizeof(Nameval));
            nvtab.nval--;
            return 1;
        }
    }
    return 0;

    /* Note that no realloc is performed to resize the array.
     * Do you think this action is needed???
     */
}
```




```
nvtab.max  == 8
nvtab.nval == 7
```

`delname("Crazy Bob")`

```
memmove(nvtab.name + i,
  nvtab.nameval + i + 1,
  (nvtab.nval-(i+1)
    * sizeof(Nameval)
);
```

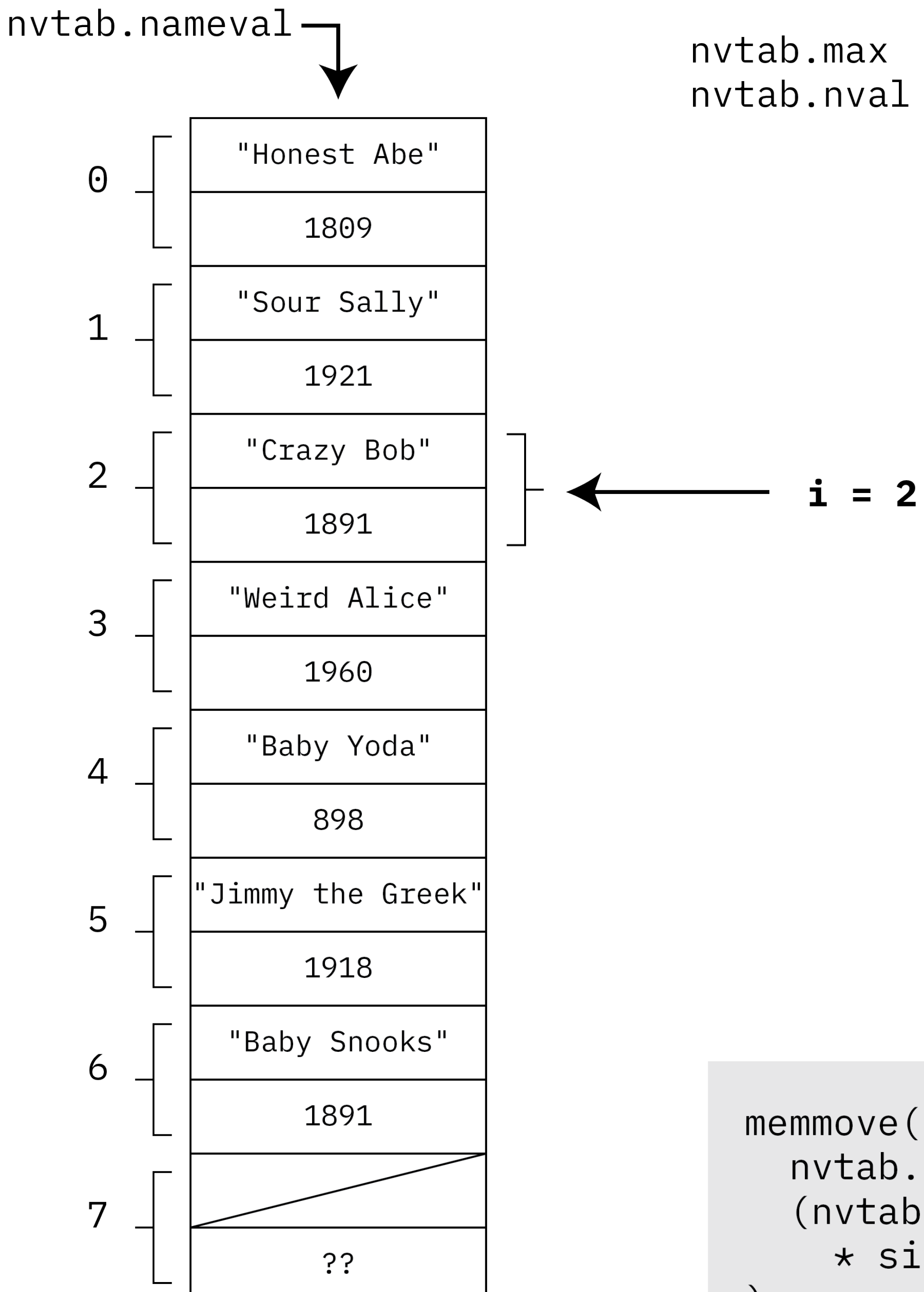



```
nvtab.max  == 8
nvtab.nval == 7
```

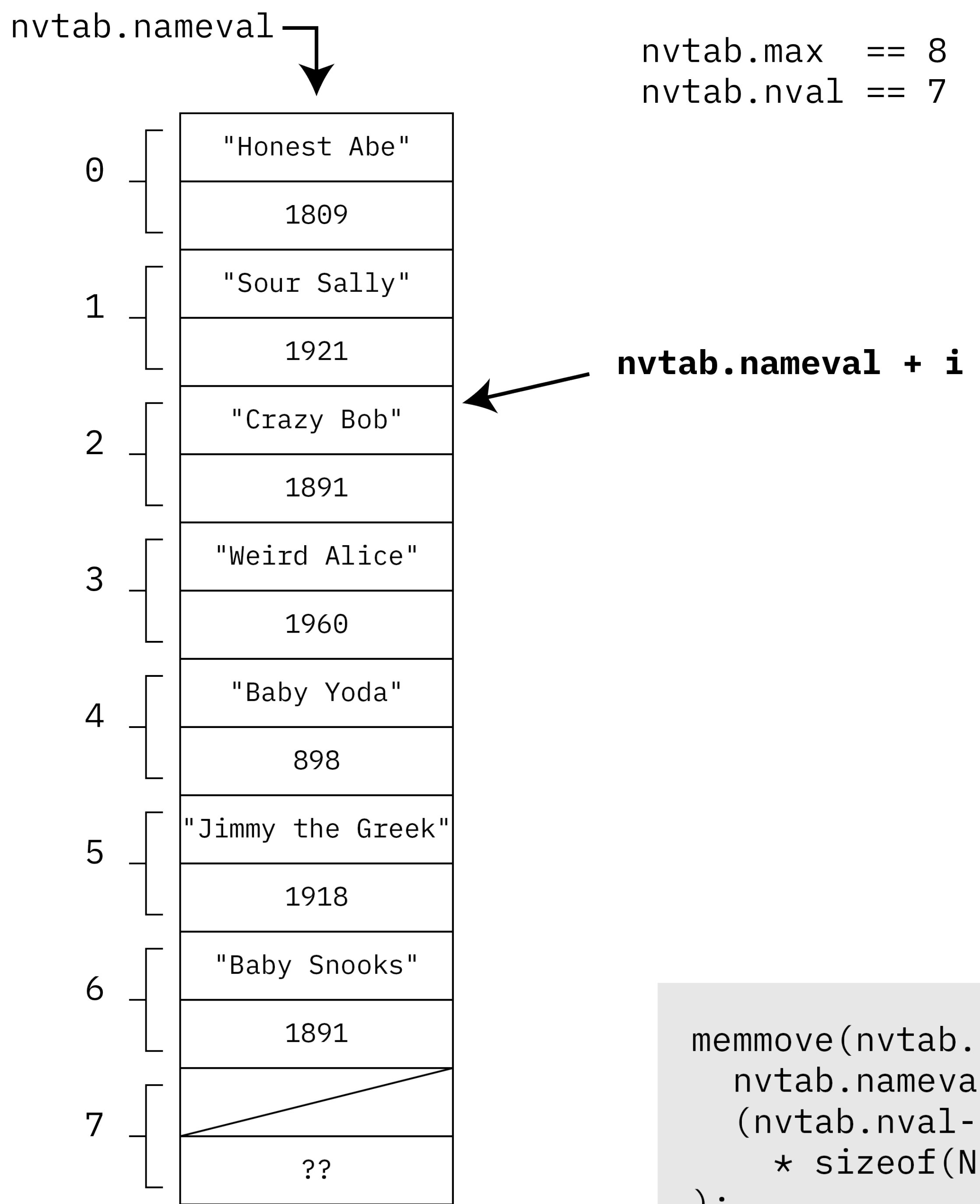
`delname("Crazy Bob")`

```
memmove(nvtab.name + i,
  nvtab.nameval + i + 1,
  (nvtab.nval-(i+1)
    * sizeof(Nameval)
);
```

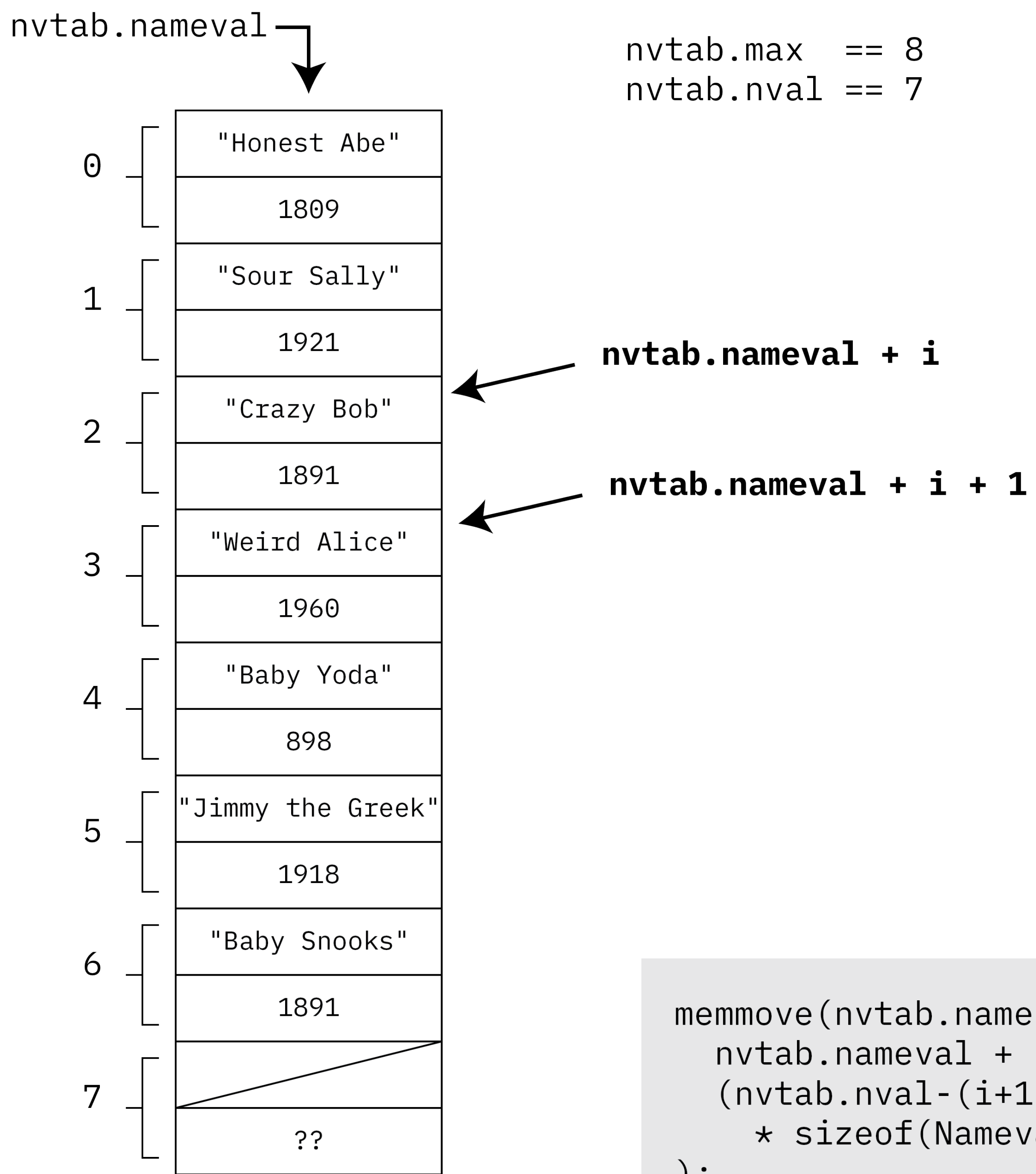
```
nvtab.max  == 8
nvtab.nval == 7
```



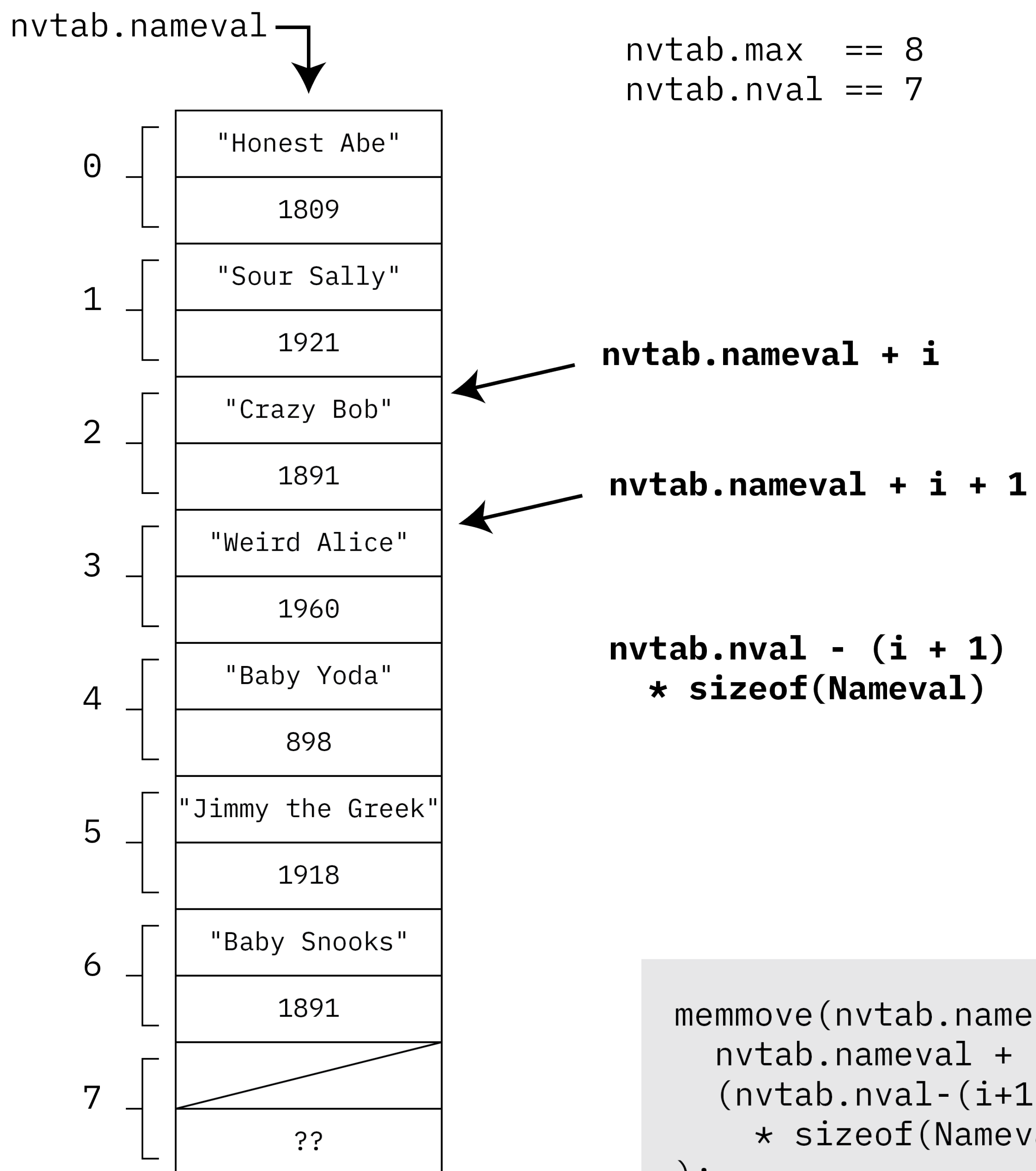

```
memmove(nvtab.name + i,
  nvtab.nameval + i + 1,
  (nvtab.nval-(i+1)
    * sizeof(Nameval)
);
```



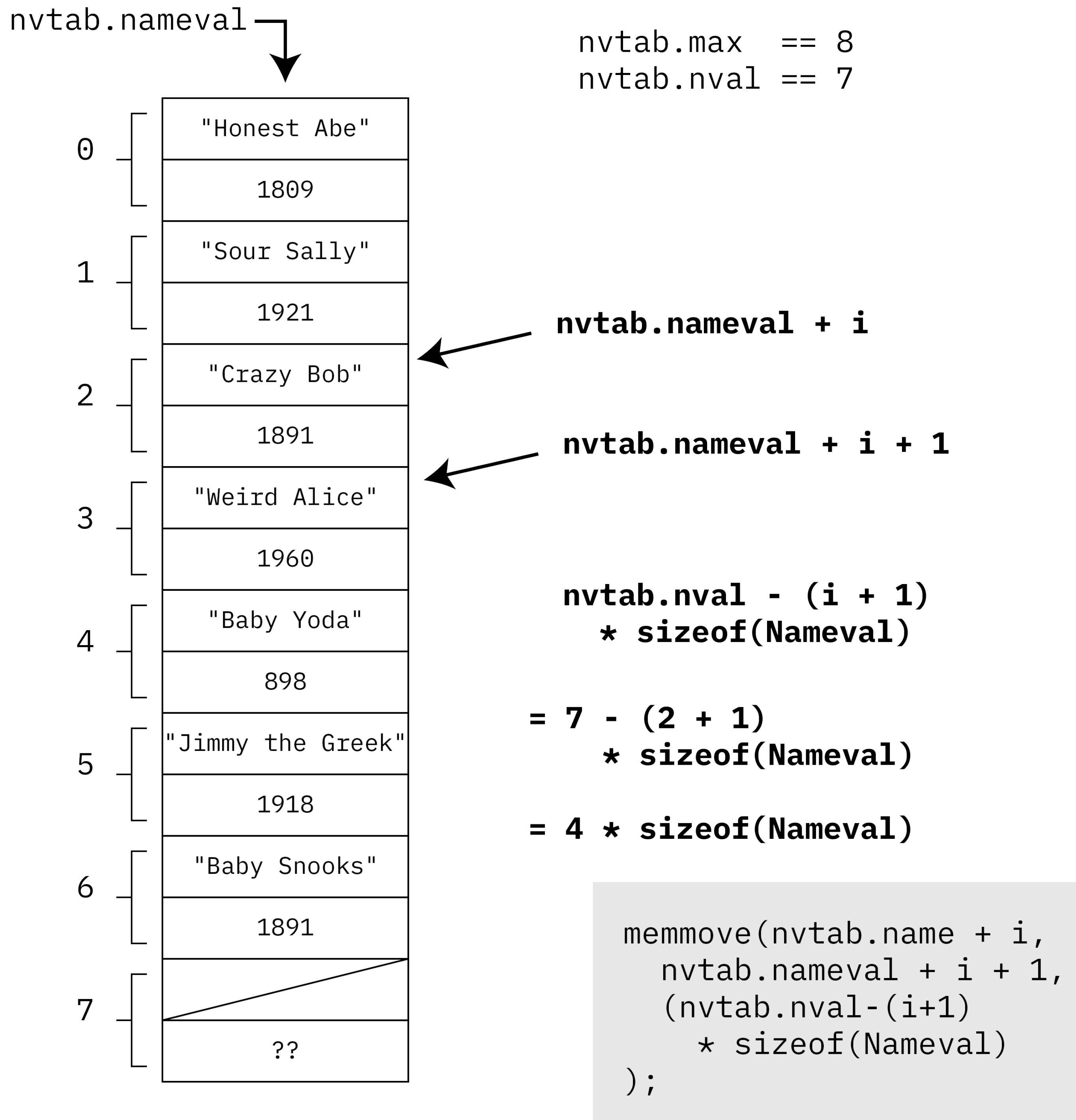

```
memmove(nvtab.name + i,
  nvtab.nameval + i + 1,
  (nvtab.nval-(i+1)
    * sizeof(Nameval)
);
```

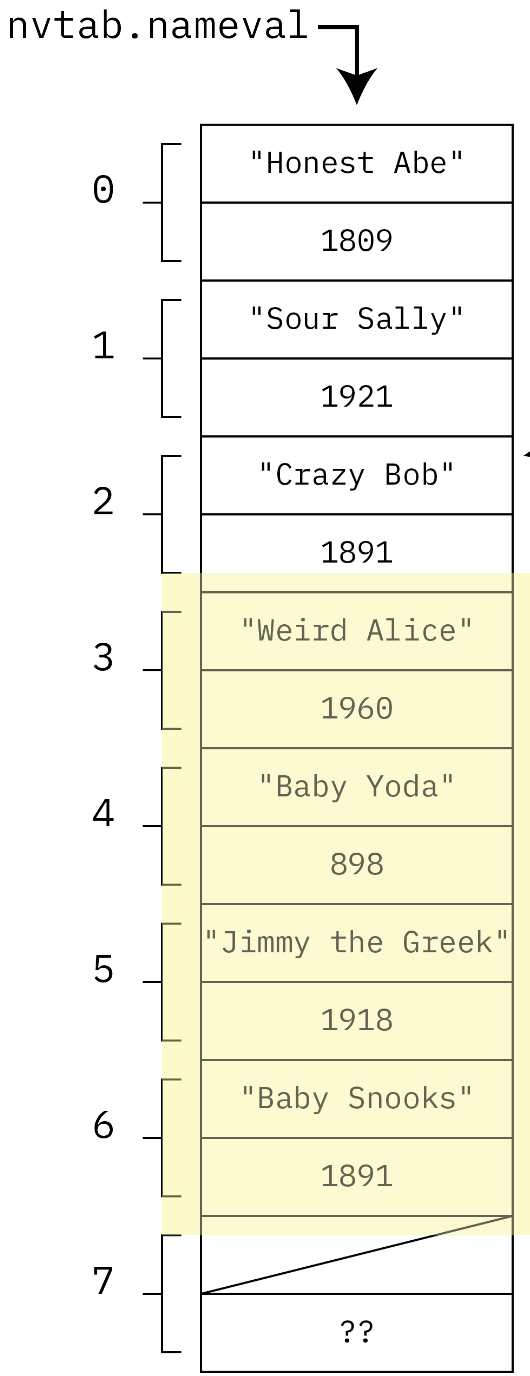



```
memmove(nvtab.name + i,
  nvtab.nameval + i + 1,
  (nvtab.nval-(i+1)
    * sizeof(Nameval)
);
```




```
memmove(nvtab.name + i,
  nvtab.nameval + i + 1,
  (nvtab.nval-(i+1)
    * sizeof(Nameval)
);
```


nvtab.nameval
nvtab.max == 8
nvtab.nval == 7
0
"Honest Abe"
1809
1
"Sour Sally"
1921
2
"Crazy Bob"
1891
3
"Weird Alice"
1960
4
"Baby Yoda"
898
5
"Jimmy the Greek"
1918
6
"Baby Snooks"
1891
7
??
nvtab.nameval + i
nvtab.nameval + i + 1
nvtab.nval - (i + 1)
* sizeof(Nameval)
= 7 - (2 + 1)
* sizeof(Nameval)
= 4 * sizeof(Nameval)
memmove(nvtab.name + i,
nvtab.nameval + i + 1,
(nvtab.nval-(i+1)
* sizeof(Nameval)
);



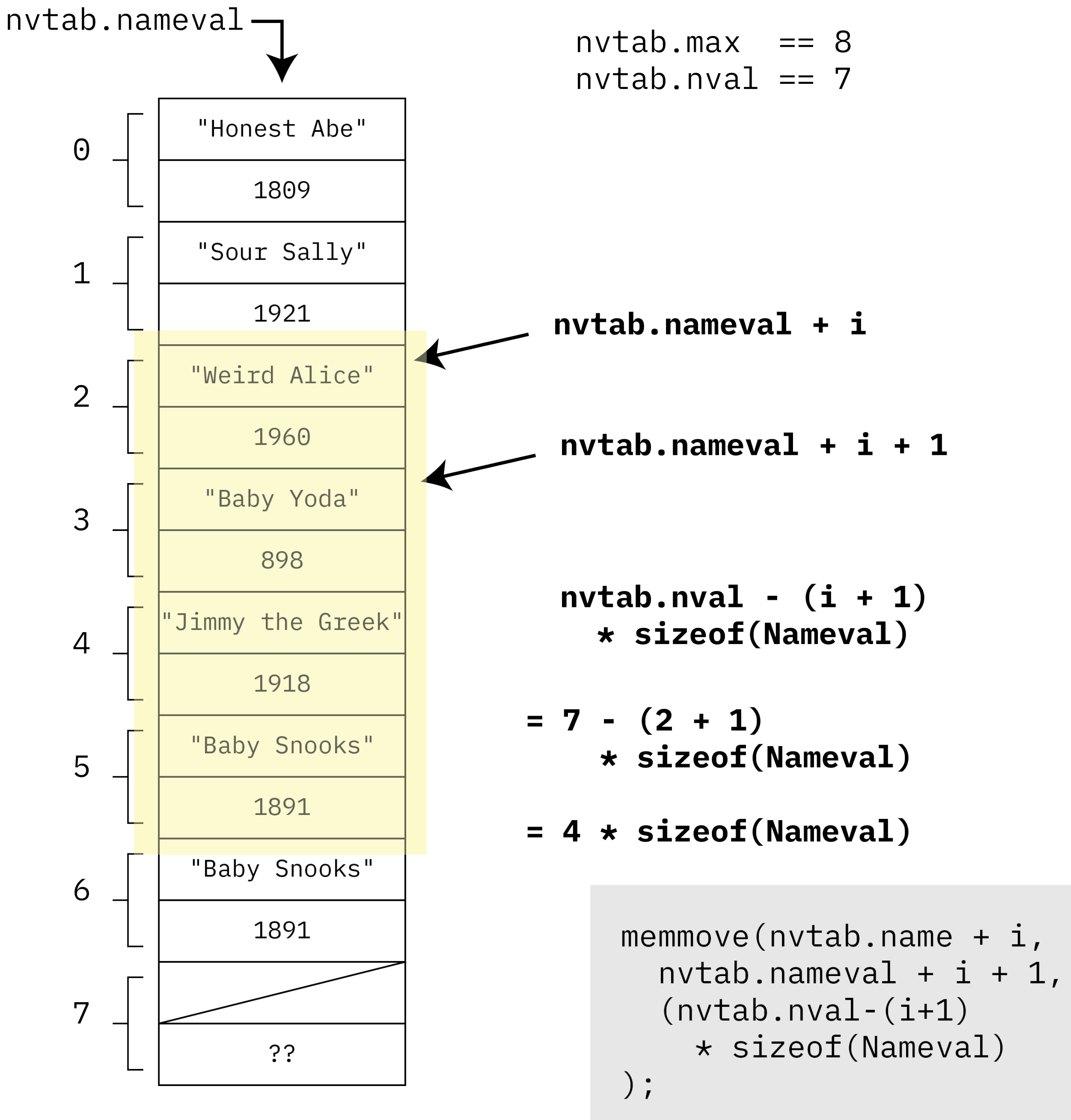

```
nvtab.max  == 8
nvtab.nval == 7
```

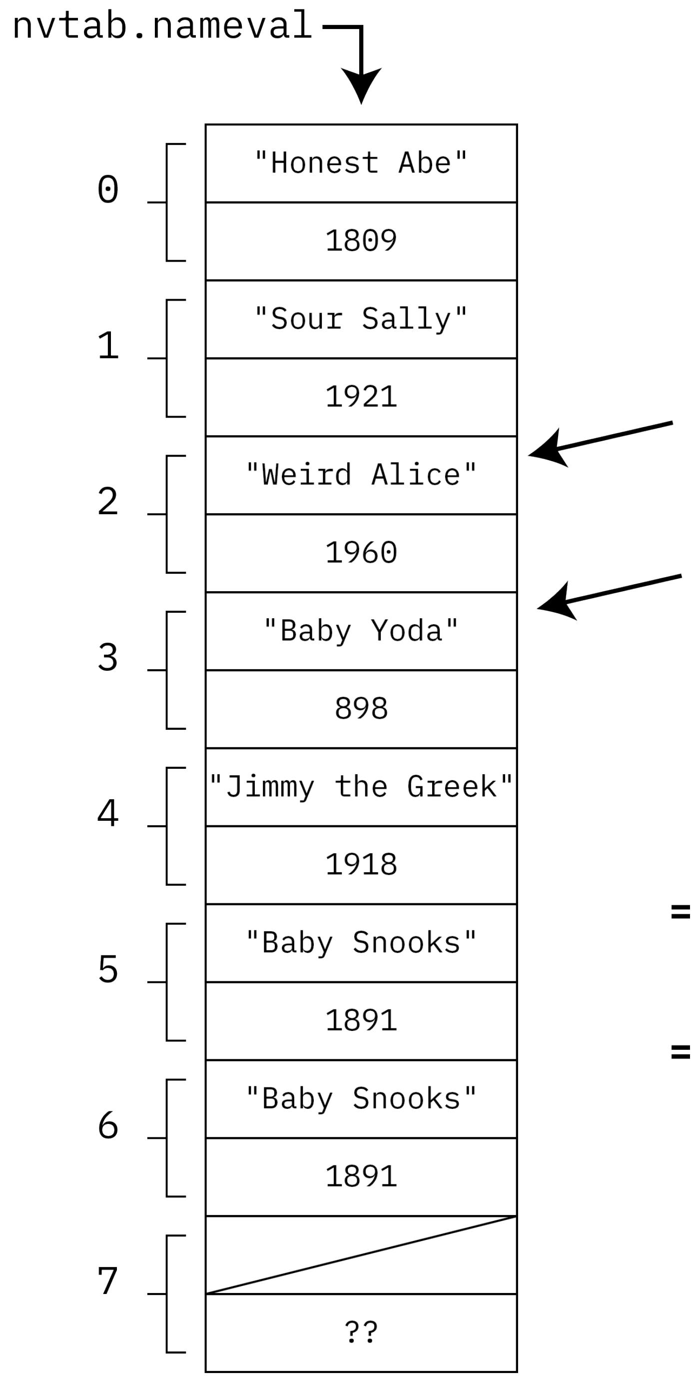
nvtab.nameval + i

nvtab.nameval + i + 1

```
nvtab.nval - (i + 1)
  * sizeof(Nameval)

= 7 - (2 + 1)
    * sizeof(Nameval)

= 4 * sizeof(Nameval)
```

```
memmove(nvtab.name + i,
  nvtab.nameval + i + 1,
  (nvtab.nval-(i+1)
    * sizeof(Nameval)
);
```


nvtab.nameval
nvtab.max == 8
nvtab.nval == 7
0
"Honest Abe"
1809
1
"Sour Sally"
1921
2
"Weird Alice"
1960
3
"Baby Yoda"
898
4
"Jimmy the Greek"
1918
5
"Baby Snooks"
1891
6
"Baby Snooks"
1891
7
??
nvtab.nameval + i
nvtab.nameval + i + 1
nvtab.nval - (i + 1)
* sizeof(Nameval)
= 7 - (2 + 1)
* sizeof(Nameval)
= 4 * sizeof(Nameval)
memmove(nvtab.name + i,
nvtab.nameval + i + 1,
(nvtab.nval-(i+1)
* sizeof(Nameval)
);




```
nvtab.max  == 8
nvtab.nval == 6
```

nvtab.nameval + i

nvtab.nameval + i + 1

```
nvtab.nval - (i + 1)
  * sizeof(Nameval)

= 7 - (2 + 1)
    * sizeof(Nameval)

= 4 * sizeof(Nameval)
```

```
memmove(nvtab.name + i,
  nvtab.nameval + i + 1,
  (nvtab.nval-(i+1)
    * sizeof(Nameval)
);
```

Chicken-and-egg...

- Consider this statement:
  - We must write our code **to be flexible for as many situations as possible**...
  - ... although this means we **cannot make some assumptions about input sizes**.
- Example:
  - For a file that processes text files, cannot make assumptions about the length of an input line
- Practical result:
  - Must (somehow) use malloc, realloc and possibly free appropriately
  - Safe alternative: getline()

getline() solution

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE *fp;
   char *line = NULL;
   size_t len = 0;   /* size_t is really just an unsigned int */
   ssize_t read;     /* ssize_t is where a function may return a size or a
                      * negative number. The first "s" means "signed".*/

   fp = fopen("/etc/motd", "r");
   if (fp == NULL) {
      exit(1);
   }

   while ((read = getline(&line, &len, fp)) != -1) {
      printf("Retrieved line of length %zu :\n", read);
      printf("%s", line);
   }

   if (line) {
      free(line);
   }
   exit(0);
}
```