Supplement to *C Interfaces and Implementations* by David R. Hanson

David R. Hanson and Norman Ramsey

September 2011

Preface

For several years I have taught from Dave Hanson's *C Interfaces and Implementations*. Hanson's interfaces provide an invaluable leg up to the student programmer, and they have enabled my students to do more ambitious projects than would be possible otherwise. But my students have consistently had difficulty with the `Array` interface. The central issue is an old one: where is the memory allocated? Unlike the other container abstractions in the book, the `Array` abstraction allocates and manages its own memory. In the terminology of the compiler writer, `Array` elements are *unboxed*.

Unboxing the elements changes the abstraction just a little, and the change is enough to warrant a slightly different interface. I hope this revision of the `Array` chapter will help students work more effectively with unboxed arrays. I have also thrown in some advice about how to work idiomatically with C-style polymorphism.

Norman Ramsey
Medford, Mass.
September 2011

Chapter 21

Unboxed Dynamic Arrays

An array is a homogeneous sequence of values in which the elements in the sequence are associated one-to-one with indices in a contiguous range. Arrays appear as built-in data types in virtually all programming languages. In some languages, like C, all array indices have the same lower bounds, and in other languages, like Modula-3, each array can have its own bounds. In C, all arrays have indices that start at zero.

Array sizes are specified at either compile time or run time. The sizes of *static* arrays are known at compile time. In ANSI C, for example, declared arrays must have sizes known at compile time; that is, in the declaration `int a[n]`, `n` must be a constant expression. A static array may be allocated at run time; for example, local arrays are allocated at run time when the function in which they appear is called, but their sizes are known at compile time.

The arrays returned by functions like `Table_toArray` are *dynamic* arrays because space for them is allocated by calling `malloc` or an equivalent allocation function. Their sizes can be determined at run time. Some languages, such as Modula-3, have linguistic support for dynamic arrays. In C, however, they must be constructed explicitly as illustrated by functions like `Table_toArray`.

The various `toArray` functions show just how useful dynamic arrays are; the `UArray` ADT described in this chapter provides a similar but more general facility. It exports functions that allocate and deallocate dynamic arrays, access them with bounds checks, and expand or contract them to hold more or fewer elements. And an element of a `UArray_T` need not be a pointer.

This chapter also describes the `UArrayRep` interface. It reveals the representation of dynamic, unboxed arrays for those few clients that need more efficient access to the array elements. Together, `UArray` and `UArrayRep` illustrate a two-level interface or a layered interface. `UArray` specifies a high-level view of an array ADT, and `UArrayRep` specifies another, more detailed view of the ADT at a lower level. The advantage of this organization is that importing `UArrayRep` clearly identifies those clients that depend on the representation of dynamic arrays. Changes to the representation thus affect only those clients, not the clients that import only `UArray`.

21.1 Boxed and unboxed

In C programs, memory is managed explicitly, and every interface must specify who is responsible for allocating and deallocating memory. The designer of any "container" type has to decide whether the objects contained in it are to be stored in *boxed* or *unboxed* form. A container that holds boxed objects stores only *pointers* to objects that are allocated elsewhere. The storage for the data itself (the "box") is under the control of the *client*. Boxed data makes for simple interfaces but higher overheads. Most containers in this book, including the `List`, `Table`, and `Set` interfaces from earlier chapters, contain boxed objects.

A container that holds *unboxed* objects allocates and manages memory for its contents. Like the `Array` interface in Chapter 10, the `UArray` interface described in this chapter stores objects in unboxed form; the memory that holds the elements is part of the `UArray` data structure.

Decisions about boxing should affect interfaces: if one container holds pointers to boxed objects and another container holds the unboxed objects themselves, the interfaces to the two containers should look different. The different interfaces should reflect the different ways in which the client and the abstraction regard memory management. Some of these differences are highlighted in Table 21.1 on page 523.

The `Array` abstraction in Chapter 10 holds *unboxed* elements, but the parts of the interface used to gain access to elements look too much like the interfaces for the other containers in the book, which hold *boxed* elements. This chapter presents a new interface, `UArray`, pronounced "unboxed array," which is a better fit for the abstraction. Think of it as a "reboot" of Chapter 10.

21.2 Interfaces

The `UArray` ADT, like other ADTs in this book, is represented as a pointer to an incomplete `struct`. It is exported by the header file `uarray.h`:

522 ⟨*uarray.h* 522⟩≡

```
#ifndef UARRAY_INCLUDED
#define UARRAY_INCLUDED

#define T UArray_T
typedef struct T *T;

⟨exported functions 524a⟩

#undef T
#endif
```

Defines:
`T`, used in chunks 524 and 531–33.
`UARRAY_INCLUDED`, never used.

Containers of boxed elements (`List`, `Table`, `Set`, `Seq`, ...)	**Containers of unboxed elements** (`UArray`)
Each element is a pointer.	An element may be a value of any type, including `struct`.
Pointees are stored outside the container.	Pointees are part of the container.
Memory for each pointee is allocated by the client.	Memory for all pointees is allocated by the container when the container is created.
The container doesn't know or care how big a pointee is.	To allocate, the container has to be told how big each pointee is.
Contained objects may outlive the container.	When a container dies, its contents die.
Changes in the container don't move pointees.	Resizing the container could move pointees.
Clients own all pointers. Clients put pointers in and take them out using functions named `get` and `put`.	The container owns all pointers. Clients borrow them temporarily (between `resize` calls) using a function named `at`.
If the container is resized, pointers previously stored in the container are still valid.	If the container is resized, old pointers to objects inside the container are invalidated.
The interface is simple, but the client has to know exactly when to allocate and free each object in the container, as well as the container itself. Overhead for memory management could be high.	The interface is less simple, but the client only has to worry about when to allocate and free the the container—all the objects in the container come along for the ride. Overhead for memory management is low.

Table 21.1: Differences between container types with boxed and unboxed elements (A "pointee" is an object pointed to, i.e., an element contained.)

The UArray ADT exports functions that operate on an array of N elements accessed by indices zero through $N-1$. In any one array, each element has the same size, but different arrays can have elements of different sizes. UArray_Ts are allocated and deallocated by

524a ⟨*exported functions* 524a⟩≡ (522) 524b ▷
```
extern T    UArray_new (int length, int size);
extern void UArray_free(T *uarray);
```
Uses T 522 522 525 530, UArray_free 532a, and UArray_new 531a.

UArray_new allocates, initializes, and returns a new array of length elements with bounds zero through length $-$ 1, unless length is zero, in which case the array has no elements. Each element occupies size bytes. The bytes in each element are initialized to zero. The size parameter must include any padding that may be required for alignment, so that when length is positive, the actual array can be created by allocating length $\cdot$ size bytes. It is a checked runtime error for length to be negative or for size to be nonpositive, and UArray_new can raise Mem_Failed.

UArray_free deallocates and clears *uarray. It is a checked runtime error for uarray or *uarray to be null.

Unlike most of the other ADTs in this book, in which all values are boxed and pointer to by void pointers, the UArray interface places no restrictions on the values of the elements; each element is just a sequence of size bytes. The rationale for this design is that UArray_Ts are used most often to build other ADTs, such as the sequences described in Chapter 11.

The functions

524b ⟨*exported functions* 524a⟩+≡ (522) ◁ 524a 524c ▷
```
extern int UArray_length(T uarray);
extern int UArray_size  (T uarray);
```
Uses T 522 522 525 530, UArray_length 532c, and UArray_size 532c.

return the number of elements in uarray and their size.

Access to an array element is provided by

524c ⟨*exported functions* 524a⟩+≡ (522) ◁ 524b 524d ▷
```
void *UArray_at(T uarray, int i);
```
Defines:
UArray_at, used in chunks 528a and 529a.
Uses T 522 522 525 530.

UArray_at returns a pointer to element number i; it's analogous to &a[i], when a is a C array. Clients access the element by casting the pointer that UArray_at returns, then dereferencing the cast pointer (see Section 21.3 below).

It is a checked runtime error for i to be greater than or equal to the length of uarray. It is an unchecked runtime error to call UArray_at and then change the size of array via UArray_resize before dereferencing the pointer returned by UArray_at.

524d ⟨*exported functions* 524a⟩+≡ (522) ◁ 524c
```
extern void UArray_resize(T uarray, int length);
extern T    UArray_copy  (T uarray, int length);
```
Uses T 522 522 525 530, UArray_copy 533b, and UArray_resize 533a.

UArray_resize changes the size of array so that it holds length elements, expanding or contracting it as necessary. If resizing makes the array larger, the new elements are initialized to zeroes. Calling UArray_resize invalidates any values returned by previous calls to UArray_at. UArray_copy is similar, but returns a copy of array that holds its first length elements. If length exceeds the number of elements in array, the excess elements in the copy are initialized to zeroes. UArray_resize and UArray_copy can raise Mem_Failed.

UArray has no functions like Table_map or Table_toArray because UArray_at provides the machinery necessary to perform the equivalent operations.

It is a checked runtime error to pass a null T to any function in this interface.

The UArrayRep interface reveals that a UArray_T is represented by a pointer to a descriptor—a structure whose fields give the number of elements in the array, the size of each element, and a pointer to the storage for the array.

525 ⟨*uarrayrep.h* 525⟩≡

```
#ifndef UARRAYREP_INCLUDED
#define UARRAYREP_INCLUDED
#define T UArray_T
struct T {
    int length;   /* number of elements in 'elems', at least 0 */
    int size;     /* number of bytes in one element */
    char *elems;  /* iff length > 0, pointer to (length * size) bytes */
};
extern void UArrayRep_init(T uarray, int length,
    int size, void *elems);
#undef T
#endif
```

Defines:
T, used in chunks 524 and 531–33.
UARRAYREP_INCLUDED, never used.
Uses UArrayRep_init 531b.

Figure 10.1 in Chapter 10 shows the descriptor for an array of 100 integers returned by UArray_new(100, sizeof int) on a machine with four-byte integers. If the array has no elements, the array field is null. Array descriptors are sometimes called *dope vectors*.

Clients of UArrayRep may read the fields of a descriptor but may not write them; writing them is an unchecked runtime error. UArrayRep guarantees that if uarray is a T and if $0 \leq$ i $<$ uarray->length, then element number i is stored at address uarray->elems + i*uarray->size.

`UArrayRep` also exports `UArrayRep_init`, which initializes the fields of a `UArray_T` structure pointed to by `uarray`. The fields are set to the values of the arguments `length`, `size`, and `elems`. This function is provided so that a client can initialize a `UArray_T` that is embedded in another structure. It is a checked runtime error for `uarray` to be null, `size` to be nonpositive, `length` to be nonzero, and `elems` to be null; also for `length` to be `nonpositive` and `elems` to be nonnull. It is an unchecked runtime error to initialize a `T` structure by means other than calling `UArrayRep_init`.

21.3 Idiomatic usage of unboxed arrays

This section presents an example that illustrates the use of polymorphic, unboxed arrays. To deal with polymorphism, we copy a pointer of type `void *` into a variable of the correct pointer type. This technique applies to all polymorphic containers, whether elements are boxed or unboxed.

What distinguishes an unboxed container is that we never "put a pointer in." Client code only takes pointers out. Clients use the pointers for reading, writing, or both.

The example code assumes that we have an unboxed array of entries from the Internet Movie Database, and that we want to select only those movies that have cool titles. A title is cool if it has the word "Cowboy" or "Alien" in it.

The implementation uses atoms, lists, sequences, `Str`, and unboxed arrays. Internally, it defines a structure that represents a movie.

526 ⟨*imdb.c* 526⟩≡

```
#include <stdlib.h>

#include "assert.h"
#include "atom.h"
#include "list.h"
#include "seq.h"
#include "str.h"
#include "uarray.h"

⟨definition of struct Movie_T and Movie_T 527a⟩

⟨movie functions 527b⟩
```

Our `Movie_T` structure holds only a fraction of what you would find in the real IMDB:

527a ⟨*definition of* `struct Movie_T` *and* `Movie_T` 527a⟩≡ (526)

```
struct Movie_T {
  const char *title;     /* an atom */
  const char *director;  /* an atom */
  int year;              /* year of first release */
  List_T cast;           /* actors in the movie; element type is an atom */
};   /* invariants: all pointers except 'cast' are non-null;
        year is at least 1878 */

typedef struct Movie_T *Movie_T;
```

Defines:
`Movie_T`, used in chunks 527–29.

Our ideas of what's cool are likely to change, to instead of writing a function that searches for "Cowboy" and "Alien," we write a slightly more general function that takes an unboxed array of movies and returns a `Seq_T` containing pointers to all the movies that have cool words in the title. The cool words are passed in sequence `cool_words`.

527b ⟨*movie functions* 527b⟩≡ (526) 528b ▷

```
Seq_T /* of Movie_T */ Movie_with_title_words
    (UArray_T /* of struct Movie_T */ movies,
     Seq_T /* of Atom */ cool_words)
{
    Seq_T cool_movies = Seq_new(10);
        /* elements have type Movie_T and
           point into the internal memory of 'movies' */
    Movie_T movie;   /* points to each movie in array */
    int i, j;

    assert(sizeof(*movie) == UArray_size(movies)); /* safety check */
    ⟨for each movie in movies, if the movie has a cool word, add it to cool_movies 528a⟩
    return cool_movies;
}
```

Uses `cool_movies` 529b, `Movie_T` 527a, and `UArray_size` 532c.

The assertion about `sizeof(*movie)` does not guarantee that the `movies` array actually contains movie structures, but if the `movies` array contains something of the wrong size, the assertion will detect it.

The assertion uses `sizeof(*movie)`, not `sizeof(struct Movie_T)`. By using the name of the variable, not its type, we maintain a single point of truth about the type of `movie`, and we protect our code against future failures:

- If the name of the `movie` variable changes and we forget to change `sizeof`, the compiler will issue an error message.
- If the *type* of the `movie` variable changes, the value of `sizeof(*movie)` might change, but it should continue to do the right thing.

The loop allocates no memory and copies no data. The pointers added to `cool_movies` are valid only as long as the `movies` array is live and is not resized. Unless the `movies` array is immutable and lives forever, this memory-management strategy is pretty risky. We mitigate the risks below with function `Movie_uarray_of_seq`. For now, here is the loop:

528a ⟨*for each movie in* `movies`*, if the movie has a cool word, add it to* `cool_movies` 528a⟩≡ (527b)

```
for (i = 0; i < UArray_length(movies); i++) {
    movie = UArray_at(movies, i);
    for (j = 0; j < Seq_length(cool_words); j++) {
        if (Str_find(movie->title, 0, 1, Seq_get(cool_words, j))) {
            Seq_addhi(cool_movies, movie); /* no data is copied */
            break;
        }
    }
}
```

Uses `cool_movies` 529b, `UArray_at` 524c 532b, and `UArray_length` 532c.

Here are some things to notice:

- By assigning the result of `UArray_at()` to `movie`, we implicitly convert the `void *` result into a pointer of type `Movie_T`. This idiom resolves `void *` polymorphism and enables us to get to the title.
- No data is copied or moved. Only pointers are copied. If elements of `UArray_T` were boxed, our work would be done—all pointers would be owned by the client. But because elements of `UArray_T` are *not* boxed, we have an unhealthy relationship between the `movies` array and the `cool_movies` sequence: if `movies` changes size or is freed, `cool_movies` has a bunch of invalid pointers.

To correct the relationship between `cool_movies` and `movies`, we define a function `Movie_uarray_of_seq`. We can use it to copy the cool movies, making them independent of the original array of movies. Copying data is a trick that is commonly used to simplify memory management.

528b ⟨*movie functions* 527b⟩+≡ (526) ◁527b 529a▷

```
UArray_T Movie_uarray_of_seq(Seq_T some_movies);
  /* takes pointers from some_movies and copies all their pointees
     into a newly allocated array.  An element of the Seq_T has
     type Movie_T, but an element of the result array has type
     'struct Movie_T' */
```

Uses `Movie_T` 527a and `Movie_uarray_of_seq` 529a.

The specification of `Movie_uarray_of_seq(Seq_T some_movies)` highlights the distinction between boxed and unboxed. The sequence, which contains boxed elements, holds pointers. The array, which contains unboxed elements, holds pointees. The pointees are `structs`.

529a ⟨*movie functions* 527b⟩+≡ (526) ◁528b 529b▷

```
UArray_T Movie_uarray_of_seq(Seq_T some_movies) {
    UArray_T result;  /* element type is struct Movie_T */
    Movie_T dst, src; /* used to copy data */
    int i;

    result = UArray_new(Seq_length(some_movies), sizeof(*dst));
    for (i = 0; i < Seq_length(some_movies); i++) {
        dst  = UArray_at(result, i);    /* to be written */
        src  = Seq_get(some_movies, i); /* to be read */
        *dst = *src; /* copies data to the 'result' array */
    }
    return result;
}
```

Defines:
`Movie_uarray_of_seq`, used in chunks 528b and 529b.
Uses `Movie_T` 527a, `UArray_at` 524c 532b, and `UArray_new` 531a.

This function shows how we use `get` with a container of boxed elements and `at` with a container of unboxed elements.

Finally, we can use these functions to produce an array of cool movies. The array contains all the movie structures it needs and can be used even after the original array is destroyed.

529b ⟨*movie functions* 527b⟩+≡ (526) ◁529a

```
/* Given an array of movie structures, return a similar array
   containing copies of the original structures, but only those
   movies that have a cool word in the title */
UArray_T cool_movies(UArray_T movies) {
    Seq_T cool_words = Seq_seq((void*)Atom_string("Cowboy"),
                               (void*)Atom_string("Alien"),
                               NULL);
    Seq_T cool_movies = Movie_with_title_words(movies, cool_words);
    UArray_T result = Movie_uarray_of_seq(cool_movies);
    Seq_free(&cool_words);
    Seq_free(&cool_movies);
    return result;
}
```

Defines:
`cool_movies`, used in chunks 527b and 528a.
Uses `Movie_uarray_of_seq` 529a.

Here's a summary of what the example shows:

- All our containers are polymorphic and use `void *` pointers, but when possible we work with pointers like `movie`, which we declared to have type `Movie_T`.
- We assign the result of `UArray_at` to a `movie` pointer. We can then read or write through `movie`. Our code has one example each or reading and writing.
- When we copy just the `movie` pointer, we have to remember that the underlying pointee is part of the array. When the array dies, the pointer will be invalidated.
- If we want movies we can store indefinitely, we make a new `UArray_T` and copy data into it.
- When expecting an unboxed array of movie structures, we check to make sure that the size of a single array element is `sizeof(*movie)`.

21.4 Implementation of unboxed arrays

The implementation of `UArray` is almost identical to the implementation of `Array` in Chapter 10. A single implementation exports both the `UArray` and `UArrayRep` interfaces:

530 ⟨*uarray.c* 530⟩≡

```
#include <stdlib.h>
#include <string.h>
#include "assert.h"
#include "uarray.h"
#include "uarrayrep.h"
#include "mem.h"

#define T UArray_T
⟨functions 531a⟩
```

Defines:
`T`, used in chunks 524 and 531–33.

UArray_new allocates space for a descriptor and for the array itself if length is positive, and calls UArrayRep_init to initialize the descriptor's fields:

531a ⟨*functions* 531a⟩≡ (530) 531b ▷

```
T UArray_new(int length, int size) {
  T array;
  NEW(array);
  if (length > 0)
    UArrayRep_init(array, length, size, CALLOC(length, size));
  else
    UArrayRep_init(array, length, size, NULL);
  return array;
}
```

Defines:
UArray_new, used in chunks 524a, 529a, and 533b.
Uses T 522 522 525 530 and UArrayRep_init 531b.

UArrayRep_init is the only valid way to initialize the fields of descriptors; clients that allocate descriptors by other means must call UArrayRep_init to initialize them.

531b ⟨*functions* 531a⟩+≡ (530) ◁ 531a 532a ▷

```
void UArrayRep_init(T uarray, int length, int size, void *elems) {
  assert(uarray);
  assert((elems && length > 0) || (length == 0 && elems == NULL));
  assert(size > 0);
  uarray->length = length;
  uarray->size = size;
  if (length > 0)
    uarray->elems = elems;
  else
    uarray->elems = NULL;
}
```

Defines:
UArrayRep_init, used in chunks 525 and 531a.
Uses T 522 522 525 530.

Calling UArrayRep_init to initialize a T structure helps reduce coupling: These calls clearly identify clients that allocate descriptors themselves and thus depend on the representation. It's possible to add fields without affecting these clients as long as UArrayRep_init doesn't change. This scenario would occur, for example, if a field for an identifying serial number were added to the T structure, and this field were initialized automatically by UArrayRep_init.

UArray_free deallocates the array itself and the T structure, and clears its argument:

532a ⟨*functions* 531a⟩+≡ (530) ◁531b 532b▷

```
void UArray_free(T *uarray) {
  assert(uarray && *uarray);
  FREE((*uarray)->elems);
  FREE(*uarray);
}
```

Defines:
UArray_free, used in chunk 524a.
Uses T 522 522 525 530.

UArray_free doesn't have to check if (*uarray)->elems is null because FREE accepts null pointers.

UArray_at provides a pointer to an element of a UArray_T:

532b ⟨*functions* 531a⟩+≡ (530) ◁532a 532c▷

```
void *UArray_at(T uarray, int i) {
  assert(uarray);
  assert(i >= 0 && i < uarray->length);
  return uarray->elems + i*uarray->size;
}
```

Defines:
UArray_at, used in chunks 528a and 529a.
Uses T 522 522 525 530.

A pointer returned by UArray_at is valid until the next call of UArray_resize.

UArray_length and UArray_size return the similarly named descriptor fields:

532c ⟨*functions* 531a⟩+≡ (530) ◁532b 533a▷

```
int UArray_length(T uarray) {
  assert(uarray);
  return uarray->length;
}
int UArray_size  (T uarray) {
  assert(uarray);
  return uarray->size;
}
```

Defines:
UArray_length, used in chunks 524b and 528a.
UArray_size, used in chunks 524b and 527b.
Uses T 522 522 525 530.

Clients of UArrayRep may access these fields directly from the descriptor. UArray_resize calls Mem's RESIZE to change the number of elements in the array, and it changes the array's length field accordingly.

533a ⟨*functions* 531a⟩+≡ (530) ◁532c 533b▷

```
void UArray_resize(T uarray, int length) {
  assert(uarray);
  assert(length >= 0);
  if (length == 0)
    FREE(uarray->elems);
  else if (uarray->length == 0)
    uarray->elems = ALLOC(length*uarray->size);
  else
    RESIZE(uarray->elems, length*uarray->size);
  uarray->length = length;
}
```

Defines:
UArray_resize, used in chunk 524d.
Uses T 522 522 525 530.

Unlike with Mem's RESIZE, a new length of zero is legal, in which case the array is deallocated, and henceforth the descriptor describes an empty dynamic array.

UArray_copy is much like UArray_resize, except that it copies array's descriptor and part or all of its array:

533b ⟨*functions* 531a⟩+≡ (530) ◁533a

```
T UArray_copy(T uarray, int length) {
  T copy;
  assert(uarray);
  assert(length >= 0);
  copy = UArray_new(length, uarray->size);
  if (copy->length >= uarray->length && uarray->length > 0)
    memcpy(copy->elems, uarray->elems, uarray->length*uarray->size);
  else if (uarray->length > copy->length && copy->length > 0)
    memcpy(copy->elems, uarray->elems, copy->length*uarray->size);
  return copy;
}
```

Defines:
UArray_copy, used in chunk 524d.
Uses T 522 522 525 530 and UArray_new 531a.

21.5 Further Reading

Some languages support variants of dynamic arrays. Modula-3 (Nelson 1991), for example, permits arrays with arbitrary bounds to be created during execution, but they can't be expanded or contracted. Lists in Icon (Griswold and Griswold 1990) are like dynamic arrays that can be expanded or contracted by adding or deleting elements from either end; these are much like the sequences described in the next chapter. Icon also supports fetching sublists from a list and replacing a sublist with a list of a different size.

Compilers for very high-level languages such as Haskell and ML sometimes provide unboxed arrays for better performance (Peyton Jones and Launchbury 1991). Unboxed arrays of double-precision floating-point numbers are especially in demand.

21.6 Exercises

21.1 Design and implement an ADT for *boxed* arrays: dynamic arrays of pointers. Your ADT should provide "safe" access to the elements of these arrays via functions similar in spirit to the functions provided by `Table`. Use `UArray` or `UArray_Rep` in your implementation.